3ª edición

Fundamentos de diseño lógico y de computadoras

www.librosite.net/mano

M. Morris Mano
Charles R. Kime

PEARSON
Prentice Hall

Fundamentos de diseño lógico y de computadoras

Tercera Edición

Fundamentos de diseño lógico y de computadoras

Tercera Edición

M. MORRIS MANO
California State University, Los Ángeles

CHARLES R. KIME
University of Wisconsin, Madison

Traducción
José Antonio Herrera Camacho
Profesor Titular de Escuela Universitaria
Universidad Politécnica de Madrid

Martina Eckert
Dra. Ingeniería en Telecomunicación
Universidad Politécnica de Madrid

Beatriz Valcuende Lozano
Ingeniera Técnica en Telefonía y Transmisión de Datos
Universidad Politécnica de Madrid

Revisión técnica
José Antonio Herrera Camacho
Profesor Titular de Escuela Universitaria
Universidad Politécnica de Madrid

PEARSON
Prentice Hall

Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima • Montevideo
San Juan • San José • Santiago • São Paulo • White Plains

Datos de catalogación bibliográfica

FUNDAMENTOS DE DISEÑO LÓGICO Y DE COMPUTADORAS
Mano, M. Morris; Kime, Charles

PEARSON EDUCACIÓN, S.A., Madrid, 2005

ISBN: 978-84-832-2688-9
Materia: Electrónica 621,3

Formato 195 × 250 mm Páginas: 648

Todos los derechos reservados.
Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual *(arts. 270 y sgts. Código Penal).*

DERECHOS RESERVADOS
© 2005 por PEARSON EDUCACIÓN, S.A.
Ribera del Loira, 28
28042 Madrid (España)

FUNDAMENTOS DE DISEÑO LÓGICO Y DE COMPUTADORES
Mano, M. Morris; Kime, Charles

ISBN: 84-205-4399-3
Depósito legal: M.

PEARSON PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN, S.A.

Authorized translation from the English language edition, entitled LOGIC AND COMPUTER DESIGN FUNDAMENTALS, 3rd Edition by Mano, M. Morris; Kime, Charkes, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2004.
ISBN 0-13-140539-X
All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission form Pearson Education, Inc.

Equipo editorial:
Editor: Miguel Martín-Romo
Técnico editorial: Marta Caicoya

Equipo de producción:
Director: José Antonio Clares
Técnico: Diego Marín

Diseño de cubierta: Equipo de diseño de Pearson Educación, S.A.

Composición: COPIBOOK, S.L.

Impreso por:

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos

CONTENIDO

Prefacio xv

□ Capítulo 1 **3**

ORDENADORES DIGITALES E INFORMACIÓN 3
1-1 Computadoras digitales 4
Representación de la información 5
Estructura de una computadora 5
Más en relación con la computadora genérica 6
1-2 Sistemas numéricos 8
Números binarios 9
Números octales y hexadecimales 10
Rangos de los números 12
1-3 Operaciones aritméticas 12
Conversión de decimal a otras bases 15
1-4 Códigos decimales 17
Suma en BCD 18
Bit de paridad 19
1-5 Códigos Gray 19
1-6 Códigos alfanuméricos 21
1-7 Sumario del capítulo 24
Referencias 24
Problemas 24

□ Capítulo 2 **27**

CIRCUITOS LÓGICOS COMBINACIONALES 27
2-1 Lógica binaria y puertas 28
Lógica binaria 28
Puertas lógicas 29

2-2	Álgebra de Boole	31
	Identidades básicas del Álgebra de Boole	33
	Manipulación algebraica	35
	El complemento de una función	37
2-3	Formas canónicas	39
	Minitérminos y maxitérminos	39
	Suma de productos	42
	Producto de sumas	44
2-4	Optimización de circuitos de dos niveles	44
	Criterios de coste	45
	Mapa de dos variables	46
	Mapa de tres variables	47
	Mapa de cuatro variables	51
2-5	Manipulación del mapa	54
	Implicantes primos esenciales	54
	Implicantes primos no esenciales	56
	Optimización de producto de sumas	57
	Condiciones de indiferencia	59
2-6	Optimización de circuitos multinivel	61
2-7	Otros tipos de puertas	65
2-8	Operador y puertas OR exclusiva	69
	Función impar	70
2-9	Salidas en altas impedancia	71
2-10	Resumen del capítulo	74
	Referencias	74
	Problemas	75

□ Capítulo 3 **81**

DISEÑO LÓGICO COMBINACIONAL 81

3-1	Conceptos de diseño y automatización	82
	Diseño jerárquico	83
	Diseño *top-down*	86
	Diseño asistido por computadora	86
	Lenguaje de descripción *hardware*	87
	Síntesis lógica	88
3-2	El espacio de diseño	90
	Propiedades de las puertas	90
	Niveles de integración	90
	Tecnologías de circuitos	91
	Parámetros tecnológicos	91
	Lógica positiva y negativa	95
	Compromisos de diseño	96
3-3	Ciclo de diseño	97
3-4	Mapeado tecnológico	104
	Especificaciones de las células	105
	Librerías	105
	Técnicas de mapeado	107

3-5	Verificación	113
	Análisis lógico manual	113
	La simulación	115
3-6	Tecnologías de implementación programables	116
	Memorias de sólo lectura	119
	Array lógico programable	121
	Arrays de lógica programables	122
3-7	Sumario del capítulo	124
	Referencias	124
	Problemas	125

☐ Capítulo 4 **133**

FUNCIONES Y CIRCUITOS COMBINACIONALES		133
4-1	Circuitos combinacionales	134
4-2	Funciones lógicas básicas	134
	Asignación, transferencia y complemento	134
	Funciones de varios bits	135
	Habilitación	137
4-3	Decodificación	140
	Extensión de decodificadores	140
	Decodificadores con señal de habilitación	143
4-4	Codificación	144
	Codificador con prioridad	145
	Expansión de codificadores	146
4-5	Selección	148
	Multiplexores	147
	Expansión de multiplexores	149
	Implementaciones alternativas de selectores	150
4-6	Implementación de funciones combinacionales	152
	Empleando decodificadores	152
	Empleando multiplexores	154
	Empleando memorias de sólo lectura	157
	Usando arrays lógicos programables	159
	Usando arrays de lógica programable	162
	Empleando tablas de búsqueda	163
4-7	HDL representación para circuitos combinacionales-VHDL	165
4-8	Representación HDL de circuitos combinacionales-Verilog	172
4-9	Resumen del capítulo	178
	Referencias	179
	Problemas	179

☐ Capítulo 5 **189**

FUNCIONES Y CIRCUITOS ARITMÉTICOS		189
5-1	Circuitos combinacionales iterativos	190
5-2	Sumadores binarios	190
	Semi-sumador	191

	Sumador completo	192
	Sumador binario con acarreo serie	193
	Sumador con acarreo anticipado	194
5-3	Resta binaria	197
	Complementos	200
	Resta con complementos	200
5-4	Sumador-restador binario	202
	Números binarios con signo	203
	Suma y resta binaria con signo	204
	Overflow o desbordamiento	206
5-5	Multiplicación binaria	208
5-6	Otras funciones aritméticas	209
	Contracción o reducción	209
	Incremento	211
	Decremento	213
	Multiplicación por constantes	213
	División por constantes	213
	Relleno a ceros y extensión	214
5-7	Representación HDL-VHDL	215
	Descripción de comportamiento	217
5-8	Representaiones HDL-Verilog	216
	Descripción de comportamiento	219
5-9	Resumen del capítulo	220
	Referencias	220
	Problemas	221

□ Capítulo 6 **227**

CIRCUITOS SECUENCIALES		227
6-1	Definición de circuito secuencial	228
6-2	Latches	230
	Latches *RS* y $\overline{R}\overline{S}$	231
	Latch *D*	233
6-3	Flip-flops	235
	Flip-flop maestro-esclavo	236
	Flip-flop disparados por flanco	238
	Símbolos gráficos estándar	239
	Entradas asíncronas	241
	Tiempos de los flip-flops	242
6-4	Análisis de circuitos secuenciales	243
	Ecuaciones de entrada	243
	Tabla de estados	245
	Diagrama de estados	247
	Temporización del circuito secuencial	248
	Simulación	250
6-5	Diseño de circuitos secuenciales	252
	Procedimiento del diseño	252
	Localización de los diagramas de estados y las tablas de estados	253

	Asignación de estados	259
	Diseñando con flip-flops *D*	259
	Diseñando con estados no usados	261
	Verificación	262
6-6	Otros tipos de flip-flops	265
	Flip-flops *JK* y *T*	265
6-7	Representación HDL para circuitos secuenciales-VHDL	267
6-8	Representación de HDL para circuitos secuenciales-Verilog	275
6-9	Resumen del capítulo	281
	Referencias	281
	Problemas	282

☐ Capítulo 7 **291**

REGISTROS Y TRANSFERENCIA DE REGISTROS		291
7-1	Registros y habilitación de carga	292
	Registro con carga en paralelo	293
7-2	Transferencia de registros	295
7-3	Operaciones de transferencia de registros	297
7-4	Nota para usuarios de VHDL y Verilog	299
7-5	Microoperaciones	299
	Microoperaciones aritméticas	300
	Microoperaciones lógicas	302
	Microoperaciones de desplazamiento	304
7-6	Microoperaciones en un registro	305
	Transferencias basadas en multiplexores	305
	Registros de desplazamiento	307
	Contador asíncrono	311
	Contadores binarios síncronos	316
	Otros contadores	317
7-7	Diseño de células básicas de un registro	319
7-8	Transferencia de múltiples registros basada en buses y multiplexores	325
	Bus triestado	326
7-9	Transferencia serie y microoperaciones	328
	Suma en serie	329
7-10	Modelado en HDL de registros de desplazamiento y contadores-VHDL	331
7-11	Modelado en HDL de registros de desplazamiento y contadores-Verilog	333
7-12	Resumen del capítulo	334
	Referencias	335
	Problemas	335

☐ Capítulo 8 **343**

SECUENCIAMIENTO Y CONTROL		343
8-1	La unidad de control	344
8-2	Algoritmo de máquinas de estados	345
	Diagrama ASM	345
8-3	Ejemplos de diagramas ASM	348

	Multiplicador binario	348
8-4	Control cableado	354
	Registro de secuencia y descodificador	357
	Un flip-flop por estado	358
8-5	Representación HDL del multiplicador binario-VHDL	363
8-6	Representación HDL del multiplicador binario-Verilog	365
8-7	Control microprogramado	368
8-8	Resumen del capítulo	370
	Referencias	370
	Problemas	371

□ Capítulo 9 **377**

MEMORIAS		377
9-1	Definiciones	378
9-2	Memoria de acceso aleatorio	378
	Operaciones de lectura y escritura	380
	Temporización de las formas de onda	381
	Características de las memorias	383
9-3	Memorias integradas SRAM	383
	Selección combinada	386
9-4	Array de circuitos integrados de memoria SRAM	389
9-5	Circuitos integrados de memoria DRAM	392
	Celda DRAM	393
	Tira de un bit de memoria DRAM	394
9-6	Tipos de memoria DRAM	398
	Memoria síncrona DRAM (SDRAM)	400
	Memoria SDRAM de doble tasa de transferencia de datos (DDR SDRAM)	402
	Memoria RAMBUS© DRAM (RDRAM)	402
9-7	Arrays de circuitos integrados de memorias dinámicas RAM	404
9-8	Resumen del capítulo	404
	Referencias	405
	Problemas	405

□ Capítulo 10 **407**

FUNDAMENTOS DEL DISEÑO DE PROCESADORES		407
10-1	Introducción	408
10-2	Rutas de datos	408
10-3	Unidad aritmético-lógica	411
	Circuito aritmético	411
	Circuito lógico	414
	Unidad lógico-aritmética	415
10-4	El desplazador	416
	Barrel Shifter	417
10-5	Representación de rutas de datos	419
10-6	La palabra de control	421
10-7	Arquitectura de un sencillo procesador	426

	Arquitectura de conjunto de instrucciones	427
	Recursos de almacenamiento	427
	Formatos de la instrucción	428
	Especificación de las instrucciones	430
10-8	Control cableado de un solo ciclo	433
	Decodificador de instrucciones	435
	Ejemplo de instrucciones y programa	437
	Problemas del procesador de un solo ciclo	439
10-9	Control cableado multiciclo	441
	Diseño del control secuencial	443
10-10	Resumen del capítulo	452
	Referencias	452
	Problemas	452

☐ Capítulo 11 **459**

ARQUITECTURA DE CONJUNTO DE INSTRUCCIONES		459
11-1	Conceptos de la arquitectura de procesadores	460
	Ciclo de operación básico de un procesador	461
	Conjunto de registros	461
11-2	Direccionamiento de los operandos	462
	Instrucciones de tres direcciones	463
	Instrucciones de dos direcciones	463
	Instrucciones de una dirección	463
	Instrucciones con cero direcciones	464
	Arquitecturas de direccionamiento	465
11-3	Modos de direccionamiento	468
	Modo implícito	469
	Modo inmediato	469
	Modos registro y registro indirecto	469
	Modo de direccionamiento directo	470
	Modo de direccionamiento indirecto	471
	Modo de direccionamiento relativo	473
	Modo de direccionamiento indexado	473
	Resumen de modos de direccionamiento	473
11-4	Arquitecturas de conjunto de instrucciones	474
11-5	Instrucciones de transferencia de datos	476
	Instrucciones de manejo de pila	476
	E/S independiente versus E/S ubicada en memoria	478
11-6	Instrucciones de manipulación de datos	479
	Instrucciones aritméticas	479
	Instrucciones lógicas y de manipulación de bits	480
	Instrucciones de desplazamiento	481
11-7	Cálculos en punto flotante	483
	Operaciones aritméticas	484
	Exponente sesgado	485
	Formato estándar de los operandos	485
11-8	Instrucciones de control de programa	487

	Instrucciones de bifurcación condicional	489
	Instrucciones de llamada y retorno de subrutinas	491
11-9	Interrupciones	492
	Tipos de interrupciones	493
	Procesamiento de interrupciones externas	494
11-10	Resumen	495
	Referencias	496
	Problemas	496

☐ Capítulo 12 **503**

UNIDADES CENTRALES DE PROCESAMIENTO RISC Y CISC		503
12-1	Ruta de datos segtmentada	504
	Ejecución de microoperaciones de *pipeline*	507
12-2	Control de la ruta de datos segmentada	509
	Rendimiento y realización de un *pipeline*	511
12-3	Procesador de conjunto reducido de instrucciones	512
	Arquitectura de conjunto de instrucciones	513
	Modos de direccionamiento	516
	Organización de la ruta de datos	516
	Organización del control	519
	Conflictos de datos	520
	Control de conflictos	527
12-4	Procesadores de conjunto de instrucciones complejo	530
	Modificaciones de la ISA	533
	Modificaciones en la ruta de datos	534
	Modificaciones de la unidad de control	535
	Control microprogramado	537
	Microprograma para instrucciones complejas	539
12-5	Más sobre diseño	542
	Conceptos de CPU de alto rendimiento	542
	Recientes innovaciones arquitecturales	545
	Sistemas digitales	546
12-6	Resumen	546
	Referencias	547
	Problemas	548

☐ Capítulo 13 **551**

ENTRADA/SALIDA Y COMUNICACIONES		551
13-1	Procesadores de E/S	552
13-2	Ejemplo de periféricos	552
	Teclado	552
	Disco duro	553
	Monitores gráficos	555
	Tasas de transferencia de E/S	556
13-3	Interfaces de E/S	556
	Unidad interfaz y bus de E/S	557

	Ejemplo de interfaz de E/S	558
	Strobing	559
	Handshaking	561
13-4	Comunicación serie	562
	Transmisión asíncrona	563
	Transmisión síncrona	564
	De vuelta al teclado	564
	Un bus de E/S serie basado en paquetes	565
13-5	Modos de transferencia	569
	Ejemplo de una transferencia controlada por programa	570
	Transferencia iniciada por interrupción	571
13-6-	Prioridad en las interrupciones	572
	Prioridad *Daisy Chain*	573
	Hardware de prioridad paralela	574
13-7	Acceso directo a memoria	576
	El controlador de DMA	576
	Transferencia de DMA	578
13-8	Procesadores de E/S	579
13-9	Resumen del capítulo	582
	Referencias	582
	Problemas	583

□ Capítulo 14 **587**

SISTEMAS DE MEMORIA		587
14-1	Jerarquía de memoria	588
14-2	Localidad de referencia	590
14-3	Memoria caché	592
	Mapeado de la caché	594
	Tamaño de línea	599
	Carga de la caché	600
	Métodos de escritura	600
	Integración de conceptos	601
	Cachés de instrucciones y datos	604
	Cachés de múltiples niveles	604
14-4	Memoria virtual	605
	Tablas de páginas	606
	Translation Lookaside Buffer	609
	Memoria virtual y caché	610
14-5	Resumen del capítulo	610
	Referencias	611
	Problemas	611
	Índice	615

PREFACIO

El objeto de este texto es proporcionar una compresión de los fundamentos del diseño lógico y de los procesadores para una amplia audiencia de lectores. Muchos de los fundamentos que se presentan no han cambiado en décadas. Por otro lado, los avances que la tecnología subyacente han tenido un gran efecto en la aplicación de estos fundamentos y se ha hecho énfasis en ellos. El proceso de diseño se ha automatizado utilizando lenguajes de descripción hardware y síntesis lógica, y la búsqueda de alta velocidad y de bajo consumo han cambiado los fundamentos del diseño de los procesadores.

El contenido de esta tercera edición continúa su enfoque en los fundamentos mientras que al mismo tiempo refleja la importancia relativa de los conceptos básicos como la tecnología y la evolución del proceso de diseño. Como ejemplo, la microprogramación, cuyo uso ha declinado como principal método de diseño de unidades de control, se trata sólo como técnica de diseño de unidades de control para realizar procesadores con instrucciones complejas. Además, con el tiempo, la terminología fundamental evoluciona y, junto con ella, nuestra perspectiva de los conceptos asociados. Por ejemplo, en esta edición, las secciones sobre circuitos NAND y NOT aparecen en el contexto más amplio de la materialización tecnológica.

El texto continua proporcionando la opción a los instructores de cubrir de forma básica tanto VHDL como Verilog® u omitir los lenguajes de descripción hardware (HDL, del inglés *Hardware Description Language*). La perspectiva de cubrir aquí, en forma de introducción, es la correspondencia de los HDLs con el hardware real que representa. Esta perspectiva vital, que es crítica al escribir los HDLs para síntesis lógica, se puede perder en un tratamiento más detallado, enfocado en el lenguaje y la facilidad de su uso.

En resumen, esta edición ofrece un fuerte énfasis en los fundamentos que subyacen al diseño lógico actual utilizando lenguajes de descripción hardware, síntesis y verificación así como los cambios en el énfasis en el uso de los fundamentos del diseño de procesadores. El enfoque de los conceptos básicos y los ejercicios manuales permanece para reforzar la comprensión completa de estos conceptos como soporte principal.

Para apoyar la perspectiva de la evolución y tratar los problemas estructurales acrecentando notablemente la longitud del capítulo, esta edición ofrece una importante reorganización de los capítulos. Los Capítulos 1 al 6 del libro tratan el diseño lógico, y los Capítulos 7 al 9 tratan el

diseño de sistemas digitales. Los Capítulos 10 al 14 se enfocan directamente en el diseño de procesadores. Esta organización proporciona unos fundamentos sólidos del diseño de sistemas digitales mientras que lleva a cabo un desarrollo gradual de abajo a arriba (*bottom-up*) de los fundamentos para utilizarlos en el diseño de los procesadores desde arriba hasta abajo (*top-down*) en los últimos capítulos. Once de los 14 capítulos contienen nuevo material que no se incluyó en la segunda edición, y aproximadamente, el 50% de los problemas se han modificado o son nuevos. Hay en torno a una docena de textos complementarios disponibles en la página web del libro, que representan tanto material nuevo como material eliminado de las anteriores ediciones. A continuación siguen los resúmenes de los temas tratados en cada capítulo.

Capítulo 1—Ordenadores digitales e información, presenta los sistemas con procesador y la representación de la información, incluyendo una nueva sección sobre los Códigos Gray.

Capítulo 2—Circuitos lógicos combinacionales, trata la teoría básica y los conceptos del diseño y optimización de los circuitos con puertas. Aparece una nueva sección sobre optimización de lógica multinivel. Además del número de literales básicos, se introduce el número de entradas por puerta como un criterio de coste más preciso en el uso de circuitos multinivel.

Capítulo 3—Diseño Lógico combinacional, ofrece una visión del proceso de diseño lógico actual y trata con las características de las puertas y retardos, el uso tecnológico de puertas como la NAND, NOR, AOI, OAI, XOR y XNOR. Se tratan los detalles de los pasos en el proceso de diseño de la lógica combinacional, incluyendo el problema de formulación, optimización lógica, materialización tecnológica y la verificación. Como parte de la materialización tecnológica, este capítulo cubre básicamente las memorias ROM, PLAs y PALs. Se proporciona una introducción a los FPGAs (*Field Programmable Gate Arrays*), enfocándose en las piezas utilizadas por los estudiantes en el laboratorio, como un suplemento en la página del texto, que permite actualizar estos cambios tecnológicos durante el tiempo de vida de esta edición.

Capítulo 4—Circuitos y funciones combinacionales, cubre el diseño de subsistemas combinacionales. Se han eliminado los remanentes de la lógica MSI, según el cambio de enfoque, por 1) los fundamentos de funciones combinacionales y su realización, y 2) las técnicas de utilización y modificación de estas funciones y sus realizaciones asociadas. Este enfoque proporciona los fundamentos para tener una visión más clara del diseño de lógica estructurada y para la visualización de la lógica resultante de la síntesis de los HDLs. Además de presentar la decodificación, codificación, conversión de códigos, selección y distribución, se han presentado nuevas funciones como la habilitación y la entrada fija. Se incluyen secciones introductorias sobre Verilog, VHDL para los diversos tipos de funciones.

Capítulo 5—Funciones y circuitos aritméticos, trata de las funciones aritméticas y su realización. Además de la representación de números, la suma, la resta y la multiplicación se han presentado las funciones de incremento, decremento, rellenado, extensión y desplazamiento, y su realización. Se han incluido descripciones en Verilog y en VHDL de las funciones aritméticas.

Capítulo 6—Circuitos secuenciales, presenta el análisis y el diseño de circuitos digitales. Se tratan los *latches*, los flip-flops maestro-esclavo, los flip-flops disparados por flanco, con especial énfasis en los de tipo D. También se tratan otros tipos de flip-flops (S-R, J-K y T), usados con menor frecuencia en los diseños modernos, pero con menos énfasis, presentándose con más profundidad en un suplemento de la página web. También se proporcionan las descripciones en VHDL y en Verilog de los flip-flops y de circuitos secuenciales.

Capítulo 7—Registros y transferencia de registros, se relacionan juntos y muy cercanos al diseño de los registros y sus aplicaciones. El diseño de registro de desplazamiento y los contadores se basa en la combinación de registros con funciones y su realización, presentadas en el Capítulo 4 y 5. Solamente se presenta el contador, llamado *ripple counter*, como un concepto

totalmente nuevo. Este enfoque concuerda con la reducción del enfoque en los circuitos originales como los MSI. Una sección nueva se enfoca en el diseño de células básicas para construir registros que realizan varias operaciones. Se presentan las descripciones en Verilog y VHDL de los diversos tipos de registros.

Capítulo 8—Secuenciamiento y control, trata el diseño de la unidad de control. Una característica adicional de la representación del Algoritmo de Máquinas de Estados (ASM, *Algorithmic State Machine*) es la bifurcación en varios caminos, análoga al «case» de Verilog y VHDL. Se enfatiza el control hardware y se reduce este énfasis en el control microprogramado.

Capítulo 9—Memorias, presenta las memorias SRAM, DRAM y las bases de las memorias como sistemas. En una nueva sección se trata las memorias síncronas DRAM y las bases de las tecnologías actuales. En la página web del texto se proporcionen modelos de las memorias en Verilog y VHDL.

Capítulo 10—Fundamentos del diseño de procesadores, trata los bancos de registros, las unidades funcionales, las rutas de datos y dos procesadores sencillos. Se diseña con algo de detalle un procesador de un solo ciclo de reloj y un nuevo procesador de varios ciclos de reloj, empleando en ambos control cableado.

Capítulo 11—Arquitecturas de conjunto de instrucciones, presenta varias facetas de la arquitectura de conjunto de instrucciones. Se trata la cuenta de direcciones, los modos de direccionamiento, las arquitecturas y los tipos de instrucciones. Los modos de direccionamiento y otros aspectos se ilustran con breves conjuntos de códigos de instrucciones.

Capítulo 12—Unidades centrales de procesamiento RISC y CISC, presenta rutas de datos segmentadas (*pipeline*) y el control. Se da un procesador de conjunto de instrucciones reducido (RISC, *Reduced Instruction Set Computer*). También se presenta un nuevo procesador de conjunto de instrucciones complejo (CISC, *Complex Instruction Set Computer*). Este diseño utiliza una unidad de control microprogramado junto con un RISC de base para realizar instrucciones complejas.

Capítulo 13—Entrada/Salida y comunicaciones, trata la transferencia de datos entre la CPU, interfaces de entrada/salida y dispositivos periféricos. Se estudia un teclado, un monitor CRT y un disco duro como periféricos, y se ilustra la interfaz de un teclado. Además se tratan las comunicaciones vía serie, incluyendo el Bus Serie Universal (USB, *Universal Serial Bus*), hasta los procesadores E/S.

Capítulo 14—Sistemas de memoria, tiene un particular enfoque sobre las jerarquías de memorias. Se presenta e ilustra el concepto de localidad de referencia, considerando la relación memoria caché/principal y memoria principal/disco duro. Se proporciona una visión de los parámetros de diseño de las memorias cachés. El tratamiento de la memoria se enfoca en la paginación y en *translation lookaside buffer* que da soporte a la memoria virtual.

Además del propio texto, hay un importante material de apoyo, que se comenta a continuación.

La **página web** del texto (http://www.librosite.net/mano) se incluye el siguiente material:

1) Doce suplementos que incluyen material nuevo y el material eliminado de las anteriores ediciones.
2) Los ficheros fuentes en VHDL y Verilog de todos los ejemplos.
3) Soluciones de aproximadamente un tercio de los problemas de todos los capítulos del texto y del material suplementario.
4) Fe de erratas.
5) Transparencias en PowerPoint® de los Capítulos 1 al 9.
6) Colección de las figuras y tablas más complejas del texto.

El paquete de **herramientas de diseño** utilizado en las impresiones locales e internacionales del texto consisten en el software *ISE Student Edition* de *Xilinx®*, que cortésmente ha proporcionado sin cargo *Xilinx Inc*. Está también disponible, mediante descarga a través de *Xilinx*, la versión de demostración del simulador lógico XE de *ModelSim®* de *Model Technology Incorporated*. Estas herramientas se pueden utilizar para realizar esquemáticos y máquinas de estados, compilar y simular código VHDL, Verilog o esquemáticos, y sintetizar diseños en CPLD y FPGA, y simular el resultado de los diseños. Con la compra de hardware de bajo coste para los experimentos, estas herramientas proporcionan a los estudiantes todo lo necesario para llevar a cabo sus experimentos en CPLDs o FPGAs.

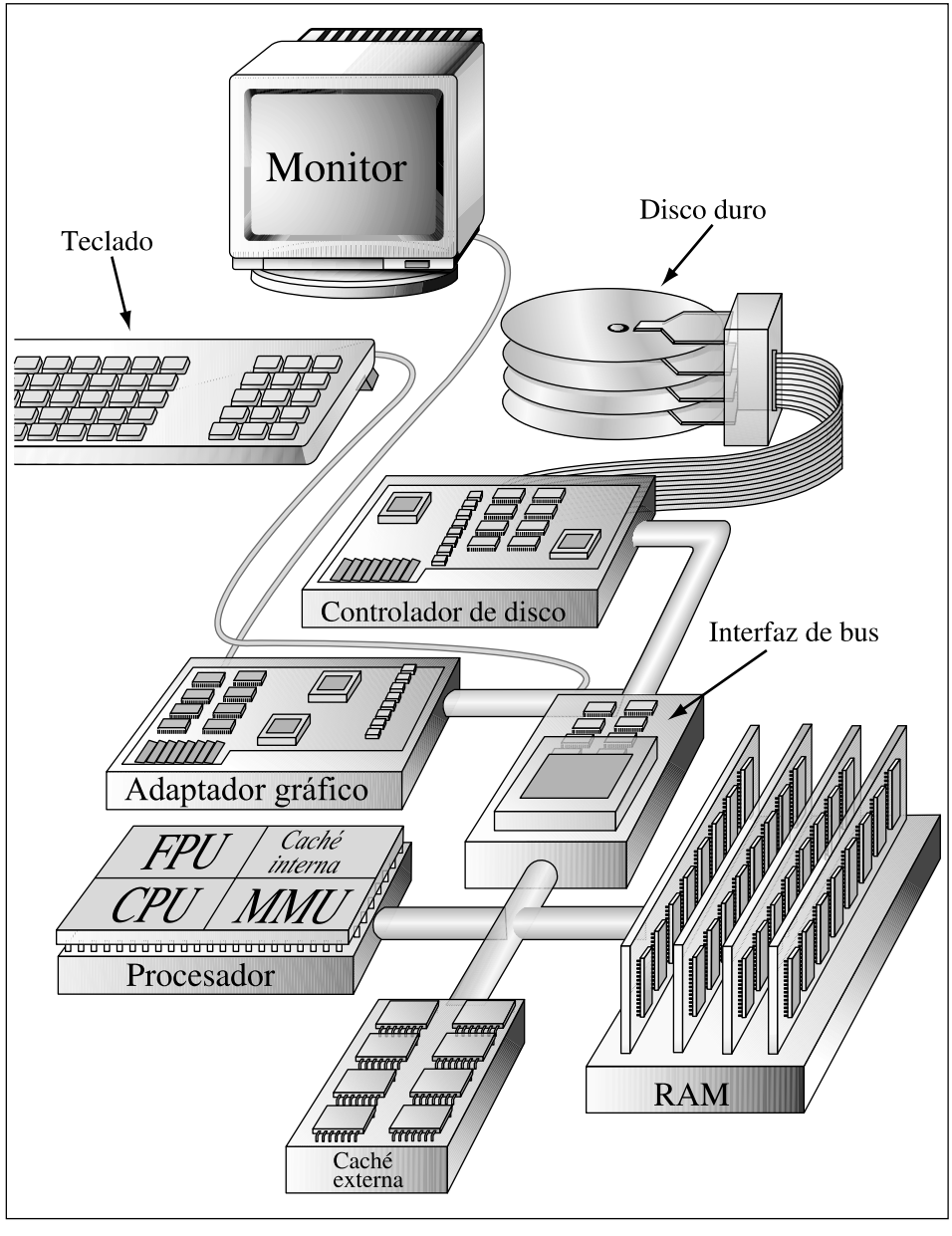
Debido a su amplio tratamiento tanto en diseño lógico y de procesadores, este libro puede servir a diferentes objetivos, desde estudiantes de cursos básicos hasta cursos de nivel superior. Los Capítulos 1 al 11, omitiendo algunas secciones, proporcionan una visión general del hardware para estudiantes de ingeniería de telecomunicación, informática, industriales o ingenierías en general, en un curso semestral. Los Capítulos del 1 al 8 dan una introducción básica al diseño lógico, que se lleva a cabo en un semestre para estudiantes de las ingenierías mencionadas. Impartir los Capítulos del 1 al 10 en un semestre proporciona un tratamiento más fuerte y actual del diseño lógico. El libro completo, impartido en dos semestres, proporciona las bases del diseño lógico y de procesadores para estos estudiantes de ingeniería. Impartir el libro completo, con el apropiado material complementario o un laboratorio podría efectuarse en una secuencia de dos semestres de un curso de diseño lógico y de procesadores. Para terminar, debido a su moderado tratamiento por pasos de un amplio espectro de temas, el libro es ideal para el autoaprendizaje de ingenieros.

Entre las diferentes contribuciones de este libro, Richard E. Haskell, Oakland University; Eugene Henry, University of Notre Dame; Sung Hu, San Francisco State University; and Walid Hubbi, New Jersey Institute of Technology proporcionaron excelentes comentarios y sugerencias sobre los dibujos de los Capítulo del 1 al 8. Su contribución a las mejoras del texto se agredeció muy sinceramente. También contribuyeron en este libro los profesores y estudiantes de la Universidad de Wisconsin. La dirección que tomó el Capítulo 12 sobre diseño de CISC se motivó por una sugerencia del Profesor Jim Smith, y el Profesor Leon Shohet sugirió mejoras específicas basadas en el uso de la 2ª edición del libro. Un agradecimiento especial para Eric Weglarz por su profunda revisión del nuevo material tanto en su contenido como en su claridad. Agradecer también a Eric y Jim Liu por la preparación de las soluciones a los problemas nuevos y a los modificados del manual del instructor. Un agradecimiento especial al equipo de Prentice Hall por sus esfuerzos en esta edición. Hay que destacar a Tom Robbins y Alice Dworkin por su dirección y apoyo, a Eric Frank por su contribución en las primeras etapas de esta edición, y a Daniel Sandin por su muy eficiente y útil manejo en la producción de esta edición.

Para terminar, un agradecimiento especial a Val Kime por su paciencia y compresión a lo largo del desarrollo de esta tercera edición.

M. MORRIS MANO
CHARLES R. KIME

FUNDAMENTOS DE DISEÑO LÓGICO Y DE COMPUTADORAS


Monitor
Teclado
Disco duro
Controlador de disco
Interfaz de bus
Adaptador gráfico
FPU
Caché interna
CPU
MMU
Procesador
RAM
Caché externa

CAPÍTULO

1

COMPUTADORAS DIGITALES E INFORMACIÓN

Los fundamentos del diseño lógico y los fundamentos del diseño de computadoras son los temas a tratar en este libro. El diseño lógico trata los conceptos básicos y las herramientas usadas en el diseño de *hardware* digital, formado por circuitos lógicos. El diseño de computadoras trata los conceptos y las herramientas adicionales usadas en el diseño de computadoras y otro tipo de *hardware* complejo. A las computadoras y al *hardware* digital se les llama, en general, sistemas digitales. Así, este libro trata del entendimiento y diseño de sistemas digitales. Debido a su generalidad y complejidad, la computadora proporciona un vehículo ideal para aprender los conceptos y las herramientas para el diseño de sistemas digitales. Además, gracias a su uso corriente, la propia computadora merece ser estudiada. Por eso, el enfoque en este libro está en las computadoras y su diseño.

La computadora no va a ser solamente un vehículo, sino también un motivo de estudio. Para ese fin, usamos el diagrama detallado de un ordenador del tipo que, normalmente, se denomina como PC (*personal computer*), de la página anterior. Usamos esa computadora genérica para destacar la importancia del material cubierto y su relación con el sistema total. Más adelante en el capítulo, discutiremos los diversos componentes principales de una computadora genérica y veremos como se relacionan con un diagrama de bloques, comúnmente usado, para describir una computadora.

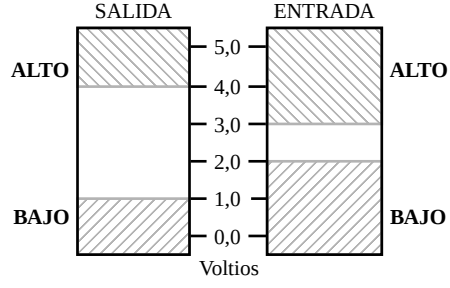
1-1 COMPUTADORAS DIGITALES

Hoy en día, las computadoras digitales tienen un papel tan prominente y creciente en la sociedad moderna, que muchas veces decimos que estamos en la «era de la información». Las computadoras están involucradas en nuestras transacciones de negocios, comunicaciones, transporte, tratamiento médico y entretenimiento. Monitorizan nuestro tiempo y medio ambiente. En el mundo industrial están fuertemente empleados en diseño, producción, distribución, y ventas. Han contribuido a muchos descubrimientos científicos y desarrollos ingenieriles que, de otra manera, hubieran sido inalcanzables. Notablemente, el diseño de un procesador para una computadora moderna no se podría hacer sin usar muchas computadoras.

La propiedad más llamativa de una computadora digital es su generalidad. Puede seguir una serie de instrucciones, llamada programa, que opera con los datos dados. El usuario puede especificar y cambiar el programa o los datos dependiendo de necesidades concretas. Como resultado de su flexibilidad, las computadoras digitales de propósito general pueden ejecutar una variedad de tareas de procesamiento de información en un espectro muy amplio de aplicaciones. La computadora de propósito general es el ejemplo más conocido de un *sistema digital*. La característica de un sistema digital es la manipulación de elementos discretos de información. Cualquier conjunto que se restrinja a un número finito de elementos contiene información discreta. Ejemplos de conjuntos discretos son los 10 dígitos decimales, las 27 letras del alfabeto, las 52 cartas de una baraja, y los 64 cuadrados de una tabla de ajedrez. Las primeras computadoras digitales se usaron principalmente para cálculos numéricos. En este caso, los elementos discretos usados fueron los dígitos. De una aplicación como ésta salió el término *computadora digital*.

Los elementos discretos de información se representan en un sistema digital por cantidades físicas llamadas *señales*. Señales eléctricas como voltajes y corrientes son las más conocidas. Los dispositivos electrónicos llamados transistores predominan en los circuitos que manejan estas señales. Las señales en la mayoría de los sistemas digitales de hoy usan justamente dos valores discretos y por eso se denominan señales *binarias*.

Típicamente representamos los dos valores discretos por rangos de valores de voltajes llamados ALTO (del término inglés HIGH) y BAJO (del término inglés LOW). Los rangos de voltios de salida y de entrada se ilustran en la Figura 1-1. El valor del voltaje de salida ALTO oscila entre 4.0 y 5.5 voltios, y el voltaje de salida BAJO entre -0.5 y 1.0 voltios. El rango de entrada mayor permite que entre 3.0 y 5.5 voltios se reconozca como ALTO, y el rango de entrada menor permite que entre -0.5 y 2.0 voltios se reconozca como BAJO. El hecho de que los rangos de entrada sean más grandes que los de salida, permite que los circuitos funcionen correctamente a pesar de variaciones en su comportamiento e indeseados voltajes de «ruido» que podrían ser añadidos o restados de las salidas.



□ **FIGURA 1-1**
Ejemplo de rangos de voltaje para señales binarias

Damos a los rangos de voltios de entrada y de salida diferentes nombres. Entre ellos están ALTO (HIGH, H) y BAJO (LOW, L), VERDAD (TRUE, T) y FALSO (FALSE, F), y 1 y 0. Está claro que los rangos de voltaje más altos están asociados con ALTO o H, y los rangos más bajos con BAJO o L. Encontramos, sin embargo, que para TRUE y 1 y FALSE y 0, hay una elección. TRUE y 1 se puede asociar o con rangos altos o bajos de voltaje y FALSE y 0 con los otros rangos. Si no se indica otra cosa, asumimos que TRUE y 1 están asociados con los rangos más altos de voltaje, H, y que FALSE y 0 están asociados con los rangos bajos, L.

¿Por qué se usa binario? En contraposición a la situación en la Figura 1-1, considere un sistema con 10 valores, que representan a los dígitos decimales. En un sistema semejante, los voltajes disponibles —es decir entre 0 y 5.0 voltios— se podrían dividir en 10 rangos, cada uno del tamaño de 0.5 voltios. Un circuito proporcionaría un voltaje de salida dentro de estos 10 rangos. Una entrada de un circuito necesitaría determinar en cual de estos 10 rangos está situado un voltaje aplicado. Si queremos permitir ruido en los voltajes, los voltajes de salida podrían oscilar en menos de 0.25 voltios para una representación de un digito dado, y los márgenes entre entradas podrían variar solamente en menos de 0.25 voltios. Esto requeriría circuitos electrónicos complejos y costosos y todavía podrían ser perturbados por pequeños voltajes de «ruido» o pequeñas variaciones en los circuitos debidas a la fabricación o el uso. Como consecuencia, el uso de estos circuitos multivalores es muy limitado. En cambio, se usan circuitos binarios donde se pueden lograr operaciones correctas de los circuitos con variaciones significativas tanto en los dos voltajes de entrada como de salida. El circuito resultante con transistores con una salida ALTA o BAJA es sencillo, fácil de diseñar y extremadamente fiable.

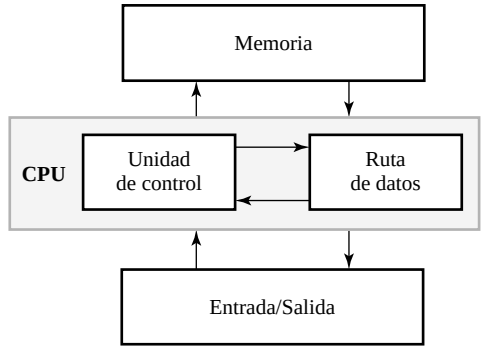
Representación de la información

Ya que 0 y 1 están asociados con el sistema de numeración binario, son los nombres preferidos para el rango de las señales. A un dígito binario se le llama *bit*. La información está representada en computadoras digitales por grupos de *bits*. Usando diferentes técnicas de codificación, se pueden construir grupos de *bits* no solamente para representar números binarios sino también otros grupos de símbolos discretos. Los grupos de *bits*, adecuadamente ordenados, pueden especificar incluso instrucciones para la computadora y datos para procesar.

Las cantidades discretas de información surgen de la naturaleza de los datos a procesar o podrían ser cuantificados intencionadamente en valores continuos. Por ejemplo, un plan de pago de nóminas tiene inherentemente datos discretos que contiene nombres de empleados, números de seguridad social, salarios semanales, tasas de ingreso, etc. Un cheque de pago de un empleado está procesado usando valores de datos discretos como las letras del alfabeto (para los nombres de los empleados), dígitos (para el sueldo) y símbolos especiales como $. En cambio, un ingeniero podría medir la velocidad de rotación de una rueda de un coche, que varía continuamente en el tiempo, pero podría grabar solamente valores específicos en forma tabular. De este modo, el ingeniero cuantifica los datos continuos, convirtiendo cada número de la tabla en una cantidad discreta de información. En un caso como éste, si la medición puede ser convertida en una señal electrónica, la cuantificación de la señal, tanto en valor y tiempo, puede ser realizada automáticamente con un dispositivo de conversión analógico-digital.

Estructura de una computadora

En la Figura 1-2 se muestra un diagrama de bloques de una computadora digital. La memoria guarda tanto programas como datos de entrada, salida e intermedios. La ruta de datos ejecuta



□ **FIGURA 1-2**

operaciones aritméticas y de otro tipo como se especifica en el programa. La unidad de control supervisa el flujo de información entre las diferentes unidades. Una ruta de datos, cuando está combinada con una unidad de control, forma un componente llamado CPU (unidad central de proceso, en inglés *central processing unit*).

El programa y los datos preparados por el usuario se transfieren a la memoria mediante un dispositivo de entrada como es el teclado. Un dispositivo de salida, como es un monitor CRT (tubo de rayos catódicos, en inglés *cathode-ray tube*) visualiza los resultados de los cálculos y los presenta al usuario. Una computadora digital puede alojar muchos dispositivos de entrada y salida diferentes, como un disco duro, *floppy*, CD-ROM y escáner. Éstos dispositivos tienen alguna lógica digital, pero muchas veces incluyen circuitos electrónicos analógicos, sensores ópticos, CRTs o LCDs (*liquid crystal displays*), y componentes electromecánicos.

La unidad de control de la CPU recupera las instrucciones, de una en una, del programa guardado en la memoria. En cada instrucción, la unidad de control manipula la ruta de datos para ejecutar la operación especificada por la instrucción. Ambos, programa y datos, están guardados en la memoria. Una computadora digital es un sistema muy potente. Puede realizar cálculos aritméticos, manipular cadenas de caracteres alfabéticos y ser programado para tomar decisiones basadas en condiciones internas y externas.

Más en relación con la computadora genérica

En este punto, vamos a presentar brevemente la computadora genérica y a relacionar sus partes con el diagrama de bloques de la Figura 1-2. En la parte inferior izquierda del diagrama, al principio de este capítulo, está el corazón de la computadora, un circuito integrado llamado el procesador. Los procesadores modernos como este son bastante complejos y se componen de millones de transistores. El procesador contiene cuatro módulos funcionales: la CPU, la FPU, la MMU, y la cache interna.

Ya hemos presentado la CPU. La FPU (unidad de punto flotante, en inglés *floating-point unit*) es parecida a la CPU, excepto que su ruta de datos y unidad de control están específicamente diseñados para realizar operaciones en punto flotante. En esencia, esas operaciones procesan información representada en forma de notación científica (por ejemplo 1.234×10^7), permitiendo a la computadora genérica manejar números muy grandes y muy pequeños. La CPU y la FPU, en relación con la Figura 1-2, contienen cada una, una ruta de datos y una unidad de control.

La MMU es la unidad de administración de la memoria. La MMU más la *cache* interna y los otros bloques, en la parte baja de la figura, etiquetados como «*Cache* Externa» y «RAM»

(*random access memory*) son todas partes de la memoria de la Figura 1-2. Las dos *caches* son un tipo especial de memoria que permite a la CPU y FPU acceder a los datos a procesar más rápidamente que sólo con la RAM. La RAM es la que se refiere generalmente como memoria. Como función principal, la MMU hace que la memoria que parece estar disponible es mucha, mucho más grande que el tamaño actual de la RAM. Esto se logra mediante traslados de datos entre la RAM y el disco duro, mostrado en la parte superior de la imagen de la computadora genérica. Así el disco duro, que estudiaremos más tarde como dispositivo de entrada/salida, aparece conceptualmente como una parte de la memoria y de entrada/salida.

Las rutas de conexión mostradas entre el procesador, la memoria y *cache* externa, son los caminos entre circuitos integrados. Típicamente se realizan con finos conductores de cobre en una placa de circuito impreso. A los caminos de conexión debajo de la interfaz del bus se le llama bus del procesador. A las conexiones encima de la interfaz del bus se le llama bus de entrada/salida (E/S). El bus del procesador y el bus E/S ligados al interfaz de buses llevan datos con diferentes números de bits y tienen diferentes maneras de controlar el movimiento de los datos. También pueden operar a diferentes velocidades. El *hardware* del interfaz de buses maneja esas diferencias de manera que los datos pueden comunicarse entre los dos buses.

El resto de estructuras de la computadora genérica se consideran parte de la E/S de la Figura 1-2. En términos de volumen, estas estructuras son las que más ocupan. Para introducir información en la computadora, se proporciona un teclado. Para ver la salida en forma de texto o gráficos, se utiliza una tarjeta con un adaptador gráfico y un monitor CRT. El disco duro, presentado previamente, es un dispositivo de almacenaje magnético electromecánico. Guarda grandes cantidades de información en forma de flujo magnético en discos giratorios cubiertos de una capa de materiales magnéticos. Para controlar el disco duro y transferir información hacia y desde él, se usa un controlador de disco. El teclado, la tarjeta de adaptador gráfico y la tarjeta de controlador de disco están todos vinculados con el bus E/S. Esto permite a estos dispositivos comunicarse mediante la interfaz de bus con la CPU y otros circuitos conectados a los buses del procesador. La computadora genérica está formada básicamente por una interconexión de módulos digitales. Para entender la operación de cada módulo, es necesario tener un conocimiento básico de los sistemas digitales y su comportamiento general. Los Capítulos 1 a 6 de este libro tratan el diseño lógico de circuitos digitales en general.

En los Capítulos 7 y 8 se presentan los componentes básicos de un sistema digital, sus operaciones y su diseño. Las características operacionales de la memoria RAM se explican en el Capítulo 9. La ruta de datos y el control de computadoras sencillas se presentan en el Capítulo 10. En los Capítulos 11 al 14 se presentan las bases del diseño de computadoras. Las instrucciones típicas empleadas en arquitecturas de conjunto de instrucciones se presentan en Capítulo 11. La arquitectura y el diseño de CPUs se examinan en el Capítulo 12. Los dispositivos de entrada y salida y los diferentes caminos con que la CPU puede comunicarse con ellos se discuten en el Capítulo 13. Finalmente, los conceptos de jerarquía de memoria relacionados con *caches* y MMU se presentan en el Capítulo 14.

Para guiar el lector por este material y para tener en mente este «bosque» examinamos minuciosamente muchos de sus «árboles», las discusiones que acompañan aparecen en las cajitas azules al principio de cada capítulo para relacionar los temas de cada capítulo con los componentes asociados en el diagrama genérico de computadoras al principio de este capítulo. Al final de nuestro viaje habremos cubierto la mayoría de los diferentes módulos de una computadora y tendremos un entendimiento de los fundamentos, que son la base del funcionamiento como del diseño.

Antes mencionamos que una computadora digital manipula elementos discretos de información y que toda la información dentro de la computadora está representada en forma binaria.

Los operandos usados en los cálculos se pueden expresar en el sistema de números binarios o en el sistema decimal por medio de un código binario. Las letras del alfabeto también se convierten a código binario. El propósito del resto de este capítulo es la introducción al sistema de numeración binario, a la aritmética binaria y de códigos binarios seleccionados como base para el estudio en los siguientes capítulos. En relación con la computadora genérica, este material es muy importante y alcanza a todos los componentes excepto algunos de E/S que involucran operaciones mecánicas y de electrónica analógica (en contraste a la digital).

1-2 SISTEMAS NUMÉRICOS

El sistema numérico decimal se emplea en la aritmética cotidiana para representar números mediante cadenas de dígitos. Dependiente de su posición en la cadena, cada dígito tiene un valor asociado a un entero como potencia en base 10. Por ejemplo, el número decimal 724.5 se interpreta de manera que representa 7 centenas, más 2 decenas, más 4 unidades y más 5 décimas. Las centenas, decenas, unidades y décimas son potencias de 10, dependiendo de la posición de los dígitos. El valor del número se calcula de la forma siguiente:

$$724.5 = 7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

La convención es escribir solamente los dígitos y deducir las potencias de 10 según su posición. En general, un número decimal con n dígitos a la izquierda del punto decimal y m dígitos a la derecha del punto decimal es representado por una cadena de coeficientes:

$$A_{n-1}A_{n-2}...A_1A_0.A_{-1}A_{-2}...A_{-m+1}A_{-m}$$

Cada coeficiente A_i es uno de los 10 dígitos (0, 1, 2, 3, 4, 5, 6, 7, 8 y 9). El valor de subíndice i determina la posición del coeficiente y, asimismo el peso $10i$ con que hay que multiplicar el coeficiente.

Al sistema numérico decimal se llama base 10, porque se multiplican los coeficientes por potencias de 10 y el sistema usa 10 dígitos diferentes. En general, un número en base r contiene r dígitos, 0, 1, 2, ..., $r - 1$, y se expresa como una potencia de r según la fórmula general

$$A_{n-1}r^{n-1} + A_{n-2}r^{n-2} + ... + A_1r^1 + A_0r^0$$
$$+ A_{-1}r^{-1} + A_{-2}r^{-2} + ... + A_{-m+1}r^{-m+1} + A_{-m}r^{-m}$$

Cuando un número se expresa en notación posicional, se escriben solamente los coeficientes y el punto de la base:

$$A_{n-1}A_{n-2}...A_1A_0.A_{-1}A_{-2}...A_{-m+1}A_{-m}$$

En general, se llama al «.» punto de base. A A_{n-1} se le llama dígito más significativo (msd, del inglés *most significant digit*) y a A_{-m} se le llama dígito menos significativo (lsd, del inglés *less significant digit*) del número. Note que si $m = 0$, el lsd es $A_{-0} = A_0$. Para distinguir entre números con bases diferentes, habitualmente se encierran los coeficientes en paréntesis y se coloca en el paréntesis derecho un subíndice para indicar la base del número. Sin embargo, si la base está clara por el contexto, no es necesario usar paréntesis. A continuación se muestra un número en base 5 con $n = 3$ y $m = 1$, y su conversión a decimal:

$$(312.4)_5 = 3 \times 5^2 + 1 \times 5^1 + 2 \times 5^0 + 4 \times 5^{-1}$$
$$= 75 + 5 + 2 + 0.8 = (82.8)_{10}$$

Note que para todos los números sin base definida, la operación se realiza con números decimales. Note también que el sistema en base 5 usa solamente cinco dígitos y, asimismo, los valores de los coeficientes de un número solamente pueden ser 0, 1, 2, 3, y 4, si se expresan en ese sistema.

Un método alternativo para la conversión a base 10, que reduce el número de operaciones, está basado en una forma factorizada de series de potencias:

$$(...((A_{n-1}r + A_{n-2})r + A_{n-3})r + ... + A_1)r + A_0$$
$$+(A_{-1} + (A_{-2} + (A_{-3} + ... + (A_{-m+2} + (A_{-m+1} + A_{-m}r^{-1})r^{-1})r^{-1}...)r^{-1})r^{-1})r^{-1}$$

Para el ejemplo de arriba,

$$(312.4)_5 = ((3 \times 5 + 1) \times 5) + 2 + 4 \times 5^{-1}$$
$$= 16 \times 5 + 2 + 0.8 = (82.8)_{10}$$

Además del sistema de numeración decimal, se usan tres sistemas de numeración a la hora de trabajar con computadoras: binario, octal, y hexadecimal. Estos están en base 2, 8, y 16 respectivamente.

Números binarios

El sistema de numeración binario es un sistema en base 2 con dos dígitos: 0 y 1. Un número binario como el 11010.11 se expresa mediante una cadena de 1 y 0 y, posiblemente, un punto binario. El número decimal equivalente a un número binario se puede encontrar mediante la expansión del número en una serie de potencias en base 2. Por ejemplo,

$$(11010)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = (26)_{10}$$

Como anteriormente se ha mencionado, los digitos de un número binario se llaman bits. Si un bit es igual a 0, no contribuye a la suma en la conversión. Por tanto, la conversión a decimal se puede obtener sumando los numeros con potencias de 2 correspondiente a los bits iguales a 1. Por ejemplo,

$$(110101.11)_2 = 32 + 16 + 4 + 1 + 0.5 + 0.25 = (53.75)_{10}$$

Los primeros 24 números obtenidos desde 2 hasta la potencia de 2 elevado a *n* se enumeran en la Tabla 1-1. Trabajando con computadoras, se refiere a 2^{10} como K (kilo), a 2^{20} como M (mega), y a 2^{30} como G (giga). Así,

$$4 \text{ K} = 2^2 \times 2^{10} = 2^{12} = 4\,096 \text{ y } 16 \text{ M} = 2^4 \times 2^{20} = 2^{24} = 16\,777\,216$$

La conversión de un número decimal a binario se puede obtener fácilmente con un método que resta sucesivamente potencias de dos al número decimal. Para convertir el número decimal *N* a binario, se busca primero el número más grande que es potencia de dos (*véase* Tabla 1-1) y que, restado de *N*, produce una diferencia positiva. Llamemos la diferencia N_1. Ahora encuentre el número más grande que es potencia de dos y que, restado de N_1, produce una diferencia positiva N_2. Continue este procedimiento hasta que la diferencia sea cero. De esta manera, el número decimal se convierte en sus componentes de potencia de dos. El número binario equivalente se obtiene de los coeficientes de una serie de potencias que forma la suma de los componentes.

□ **TABLA 1-1**
Potencias de dos

n	2^n	n	2^n	n	2^n
0	1	8	256	16	65 536
1	2	9	512	17	131 072
2	4	10	1 024	18	262 144
3	8	11	2 048	19	524 288
4	16	12	4 096	20	1 048 576
5	32	13	8 192	21	2 097 152
6	64	14	16 384	22	4 194 304
7	128	15	32 768	23	8 388 608

Los 1 aparecen en el número binario en las posiciones para los que aparecen términos en la serie de potencias, y aparecen 0 en el resto de posiciones. Este método se muestra mediante la conversión del número decimal 625 a binario de la siguiente manera:

$$\begin{aligned} 625 - 512 = 113 &= N_1 \qquad 512 = 2^9 \\ 113 - 64 = 49 &= N_2 \qquad 64 = 2^6 \\ 49 - 32 = 17 &= N_3 \qquad 32 = 2^5 \\ 17 - 16 = 1 &= N_4 \qquad 16 = 2^4 \\ 1 - 1 = 0 &= N_5 \qquad 1 = 2^0 \end{aligned}$$

$$(625)_{10} = 2^9 + 2^6 + 2^5 + 2^4 + 2^0 = (1001110001)_2$$

Números octales y hexadecimales

Como hemos mencionado anteriormente, todos las computadoras y sistemas digitales usan la representación binaria. Los sistemas de numeración octal (en base 8) y hexadecimal (en base 16) son útiles para representar cantidades binarias indirectamente porque poseen la propiedad de que sus bases son de potencia a 2. Ya que $2^3 = 8$ y $2^4 = 16$, cada dígito octal corresponde a tres dígitos binarios y cada dígito hexadecimal corresponde a cuatro dígitos binarios.

La representación más compacta de números binarios en octal o hexadecimal es mucho más conveniente para las personas que usar cadenas de bits en binario que son tres o cuatro veces más largas. Así, la mayoría de los manuales de computadoras usan números octales o hexadecimales para especificar cantidades binarias. Un grupo de 15 bits, por ejemplo, puede ser representado en el sistema octal con solamente cinco dígitos. Un grupo de 16 bits se puede representar en hexadecimal con cuatro dígitos. La elección entre una representación octal o hexadecimal de números binarios es arbitraria, aunque la hexadecimal tiende a ser la más usada, ya que los bits aparecen frecuentemente en grupos de tamaño divisible por cuatro.

El sistema de numeración octal es el sistema en base 8 con los dígitos 0, 1, 2, 3, 4, 5, 6 y 7. Un ejemplo de un número octal es 127.4. Para determinar su valor decimal equivalente, extendemos el número en una serie con base 8:

$$(127.4)_8 = 1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times 8^{-1} = (87.5)_{10}$$

Véase que los dígitos 8 y 9 no pueden aparecer en un número octal.

Es usual usar los primeros *r* dígitos del sistema decimal, empezando con 0, para representar los coeficientes en un sistema en base *r si r* es menor que 10. Las letras del alfabeto se usan para complementar los dígitos si *r* es 10 o más. El sistema numérico hexadecimal es un sistema de numeración en base 16 con los 10 primeros dígitos tomados del sistema de numeración decimal y las letras A, B, C, D, E, y F usadas para los valores 10, 11, 12, 13, 14 y 15, respectivamente. Un ejemplo de un número hexadecimal es

$$(B65F)_{16} = 11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^0 = (46687)_{10}$$

Los 16 primeros números de los sistemas de numeración decimal, binario, octal y hexadecimal se encuentran en la Tabla 1-2. Note que la secuencia de números binarios sigue un patrón preescrito. El bit menos significativo alterna entre 0 y 1, el segundo bit significativo alterna entre dos 0 y dos 1, el tercer bit significativo alterna entre cuatro 0 y cuatro 1, y el bit más significativo alterna entre ocho 0 y ocho 1.

La conversión de binario a octal se consigue fácilmente dividiendo el número binario en grupos de tres bits cada uno, empezando por punto binario y procediendo hacia la izquierda y hacia la derecha. El dígito octal correspondiente se asigna a cada grupo. El siguiente ejemplo demuestra el procedimiento:

$$(010\ 110\ 001\ 101\ 011.\ 111\ 100\ 000\ 110)_2 = (26153.7406)_8$$

El dígito octal correspondiente a cada grupo de tres bits se obtiene de las primeras ocho filas de la Tabla 1-2. Para conseguir que el número total de bits sea un múltiplo de tres, se puede añadir 0 a la izquierda de la cadena a la izquierda del punto binario. Más importante: hay que añadir 0 a la derecha de la cadena de bits a la derecha del punto binario para conseguir que el número de bits sea un múltiplo de tres y obtener el resultado octal correcto.

□ **TABLA 1-2**
Números con diferentes bases

Decimal (base 10)	Binario (base 2)	Octal (base 8)	Hexadecimal (base 16)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

La conversión de binario a hexadecimal es similar, excepto que el número binario es dividido en grupos de cuatro dígitos. El número binario anterior se convierte a hexadecimal de la siguiente manera:

$$(0010\ 1100\ 0110\ 1011.\ 1111\ 0000\ 0110)_2 = (2C6B.F06)_{16}$$

El dígito hexadecimal correspondiente para cada grupo de cuatro bits se obtiene de la Tabla 1-2.

La conversión de octal o hexadecimal a binario se consigue invirtiendo el procedimiento anterior. Cada dígito octal se convierte en su equivalente binario de 3 bits y se añade 0 adicionales. De forma parecida, cada dígito hexadecimal se convierte a su equivalente binario de 4 bits. Esto se muestra en los siguientes ejemplos:

$$(673.12)_8 = 110\ 111\ 011.\ 001\ 010 = (110111011.00101)_2$$
$$(3A6.C)_{16} = 0011\ 1010\ 0110.\ 1100 = (1110100110.11)_2$$

Rangos de los números

En las computadoras digitales, el rango de los números que se pueden representar está basado en el número de bits disponibles en la estructura del hardware que almacena y procesa la información. El número de bits en estas estructuras son normalmente potencias de dos, como 8, 16, 32 y 64. Como el número de bits está predeterminado por las estructuras, la adición de ceros al principio y al final es necesario para representar los números, así el rango de números que pueden ser representados está también predeterminado.

Por ejemplo, para una computadora que procesa enteros sin signo de 16 bits, el número 537 está representado como 0000001000011001. El rango de enteros que pueden ser manejados por esta representación va de 0 a $2^{16} - 1$, eso es de 0 a 65 535. Si la misma computadora procesa fracciones sin signo de 16 bits con el punto binario a la izquierda del dígito más significativo, entonces el número 0.375 está representado por 0.0110000000000000. El rango de fracciones que se puede representar es de 0 a $(2^{16} - 1)/2^{16}$, o de 0.0 a 0.9999847412.

En capítulos posteriores, trabajaremos con representaciones de bits fijas y rangos para números binarios con signo y números en punto flotante. En ambos casos, algunos bits se usan para representar otra información que simples valores enteros o fraccionados.

1-3 OPERACIONES ARITMÉTICAS

Las operaciones aritméticas con números en base *r* siguen las mismas leyes que los números decimales. Sin embargo, si se usa una base diferente a la muy conocida base 10, hay que tener cuidado en solamente usar los *r* dígitos permitidos y realizar todos los cálculos con dígitos en base *r*. Ejemplos para la suma de dos números binarios son los siguientes:

Acarreos:	**00000**	**101100**
Sumando:	01100	10110
Sumando:	+10001	+10111
Suma:	11101	101101

La suma de dos números binarios se calcula según las mismas reglas que para los números decimales, excepto que el dígito de la suma puede ser solamente 1 o 0. Asimismo, un acarreo en

binario aparece si la suma en alguna posición es mayor que 1 (un acarreo en decimal aparece si la suma en alguna posición es mayor que 9). El acarreo generado en una posición dada se suma a los bits de la columna siguiente más significativa. En el primer ejemplo, como todos los acarreos son 0, los bits resultantes son simplemente la suma de los bits de cada columna. En el segundo ejemplo, la suma de los bits de la segunda columna a partir de la derecha es 2, resultando un bit igual a 0 y un bit de acarreo igual 1 (2 = 2 + 0). El bit de acarreo se suma con los 1 de la tercera posición, resultando una suma igual a 3, que produce un bit resultante igual a 1 y un bit de acarreo igual a 1 (3 = 2 + 1).

El siguiente ejemplo es una resta de dos números binarios:

Acarreos:	**00000**	**00110**		**00110**
Minuendo:	10110	10110	10011	11110
Substraendo:	−10010	−10011	−11110	−10011
Diferencia:	00100	00011		−01011

Las reglas para la resta son las mismas que en decimal, excepto que un acarreo dentro de una columna dada suma 2 al bit del minuendo (un acarreo en el sistema decimal suma 10 al dígito del minuendo). En el primer ejemplo presentado no se genera acarreo, así los bits de la diferencia son simplemente los bits del minuendo menos los bits a restar. En el segundo ejemplo, en la posición derecha, el bit para restar es 1 con un bit 0 en el minuendo, así es necesario un acarreo de la segunda posición, según se ilustra. Esto da como resultado un bit de la diferencia en la primera posición de 1 (2 + 0 − 1 = 1). En la segunda posición, el acarreo está restado, así hace falta otro acarreo. Recuerde que, en el caso de que el substraendo es más grande que el minuendo, restamos el minuendo del substraendo y añadimos un signo negativo. Así es en el caso del tercer ejemplo, donde se muestra este intercambio de los dos operandos.

La última operación a presentar es la multiplicación binaria, que es bastante sencilla. Los dígitos del multiplicador son siempre 0 o 1. Por tanto, los productos parciales son iguales al multiplicando o a 0. Se demuestra la multiplicación en el siguiente ejemplo:

Multiplicando:	1011
Multiplicador:	× 101
	1011
	0000
	1011
Producto:	110111

Las operaciones aritméticas en octal, hexadecimal, o cualquier otro sistema en base *r* requieren normalmente la formulación de tablas de las que se puede obtener sumas y productos de dos dígitos en esta base. Una alternativa más sencilla para sumar dos números en base *r* es convertir cada par de dígitos de una columna a decimal, sumar los dígitos en decimal, y después convertir el resultado correspondiente a la suma y al acarreo en el sistema de numeración en base *r*. Ya que la suma se lleva a cabo en decimal, nos podemos fiar de nuestra memoria para obtener las entradas de la tabla de la suma en decimal. La secuencia de pasos para sumar los dos números hexadecimales 59F y E46 se muestra en el Ejemplo 1-1.

EJEMPLO 1-1 Adición hexadecimal

Realice la suma $(59F)_{16} + (E46)_{16}$:

Hexadecimal	Cálculo equivalente decimal				
	1		1		
5 9 F	5	Acarreo	9	15	Acarreo
E 4 6	14		4	6	
1 3 E 5	**1** 19 = 16 + **3**		14 = **E**	21 = 16 + **5**	

Las columnas de cálculo decimal equivalente a la derecha muestran el razonamiento mental que hay que llevar a cabo para producir cada dígito de la suma hexadecimal. En vez de sumar F + 6 en hexadecimal, sumamos los decimales equivalentes, 15 + 6 = 21. Después reconvertimos al hexadecimal anotando que 21 = 16 + 5. Esto da como resultado un dígito de suma de 5 y un acarreo de 1 para la columna siguiente más significativa. Las otras dos columnas se suman de manera similar. ■

La multiplicación de dos números en base a *r* se puede conseguir haciendo todas las operaciones aritméticas en decimal y convirtiendo resultados intermediarios de uno en uno. Esto se muestra con la multiplicación de dos números octales en el siguiente Ejemplo 1-2.

EJEMPLO 1-2 Multiplicación octal

Realice la multiplicación $(762)_8 \times (45)_8$:

Octal	Octal		Decimal		Octal
7 6 2	5 × 2	=	10 = 8 + 2	=	1**2**
4 5	5 × 6 + 1	=	31 = 24 + 7	=	3**7**
4 6 7 2	5 × 7 + 3	=	38 = 32 + 6	=	**46**
3 7 1 0	4 × 2	=	8 = 8 + 0	=	1**0**
4 3 7 7 2	4 × 6 + 1	=	25 = 24 + 1	=	3**1**
	4 × 7 + 3	=	31 = 24 + 7	=	**37**

Los cálculos de la derecha muestran los cálculos mentales para cada par de dígitos octales. Los dígitos octales 0 a 7 tienen el mismo valor que sus dígitos decimales correspondientes. La multiplicación de dos dígitos octales más un acarreo, derivado del cálculo de la linea anterior, se realiza en decimal, y el resultado se reconvierte en octal. El dígito izquierdo del resultado octal de dos dígitos da lugar a un acarreo que hay que sumar al producto de dígitos de la linea siguiente. Los dígitos azules del resultado octal de los cálculos decimales se copian a los productos parciales octales a la izquierda. Por ejemplo, $(5 \times 2)_8 = (12)_8$. El dígito izquierdo, 1, es el acarreo que hay que sumar al producto $(5 \times 6)_8$, y el último dígito significativo, **2**, es el dígito correspondiente del producto parcial octal. Si no hay ningún dígito del producto al cual se puede sumar el acarreo, el acarreo está escrito directamente dentro del producto octal parcial, como en el caso de **4** en **46**. ■

Conversión de decimal a otras bases

La conversión de un número en base *r* a un decimal se realiza expandiendo el número en una serie de potencias y sumando todos los términos, como se mostró anteriormente. Presentamos ahora un procedimiento general para la operación inversa de convertir un número decimal a un número en base *r* que está relacionado con la expansión alternativa a decimal en la Sección 1-2. Si el número incluye un punto decimal, es necesario separar el número en una parte entera y una parte fraccionaria, ya que hay que convertir las dos partes de diferente manera. La conversión de un entero decimal a un número en base *r* se hace dividiendo el número en todos los cocientes de *r* y acumulando los restos. Este procedimiento se explica mejor con un ejemplo.

EJEMPLO 1-3 Conversión de enteros decimales a octal

Convierta el número decimal 153 a octal:

La conversión es a base 8. Primero, se divide 153 por 8 que resulta en un cociente de 19 y un resto 1, como ilustrado en negrita. Después 19 es dividido por 8 lo que resulta en un cociente de 2 y un resto de 3. Finalmente, 2 es dividido por 8 resultando en un cociente de 0 y un resto de 2. Los coeficientes del número octal deseado se obtiene de los restos:

$$153/8 = 19 + \mathbf{1}/8 \qquad \mathbf{Resto = 1} \qquad \text{Dígito menos significativo}$$
$$19/8 = 2 + \mathbf{3}/8 \qquad \mathbf{= 3}$$
$$2/8 = 0 + \mathbf{2}/8 \qquad \mathbf{= 2} \qquad \text{Dígito más significativo}$$
$$(153)_{10} = (\mathbf{231})_8$$

■

Véase en el Ejemplo 1-3 que se lee los restos desde el último hacia el primero, como se indica mediante la flecha, para obtener el número convertido. Los cocientes se dividen por *r* hasta que el resultado sea 0. También podemos usar este procedimiento para convertir enteros decimales a binario como se muestra en el Ejemplo 1-4. En este caso, la base del número convertido es 2, y asimismo, todas las divisiones hay que hacerlas por 2.

EJEMPLO 1-4 Conversión de enteros decimales a binario

Convierta el número decimal 41 a binario:

γ

$$41/2 = 20 + \mathbf{1}/2 \qquad \mathbf{Resto = 1} \qquad \text{Dígito menos significativo}$$
$$20/2 = 10 \qquad \mathbf{= 0}$$
$$10/2 = 5 \qquad \mathbf{= 0}$$
$$5/2 = 2 + \mathbf{1}/2 \qquad \mathbf{= 1}$$
$$2/2 = 1 \qquad \mathbf{= 0}$$
$$1/2 = 0 + \mathbf{1}/2 \qquad \mathbf{= 1} \qquad \text{Dígito más significativo}$$
$$(41)_{10} = (\mathbf{101001})_2$$

Por supuesto, se puede convertir el número decimal mediante la suma de potencias de dos:

$$(41)_{10} = 32 + 8 + 1 = (101001)_2$$

■

La conversión de una fracción decimal en base *r* se logra mediante un método similar al que se usa para enteros, excepto que se usa la multiplicación por *r* en vez de la división, y se acumulan enteros en vez de restos. Otra vez, se explica el método mejor con un ejemplo.

EJEMPLO 1-5 Conversión de fracciones decimales a binario

Convierta el número decimal 0.6875 a binario:

Primero, se multiplica 0.6875 por 2 para conseguir un entero y una fracción. La nueva fracción se multiplica por 2 para conseguir otro entero y otra fracción. Se continúa este procedimiento hasta que la parte fraccionaria sea igual a 0 o hasta que haya suficientes dígitos para conseguir exactitud suficiente. Los coeficientes del número binario se obtienen de los enteros de la manera siguiente:

$0.6875 \times 2 = \mathbf{1}.3750$	**Entero = 1**	Dígito más significativo
$0.3750 \times 2 = \mathbf{0}.7500$	**= 0**	
$0.7500 \times 2 = \mathbf{1}.5000$	**= 1**	
$0.5000 \times 2 = \mathbf{1}.0000$	**= 1**	Dígito menos significativo

$(0.6875)_{10} = (0.\mathbf{1011})_2$ ■

Véase en el ejemplo anterior que se leen los enteros desde el primero hacia el último, como indica la flecha, para obtener el número convertido. En el ejemplo, aparece un número finito de dígitos en el número convertido. El proceso de multiplicación de fracciones por *r* no termina necesariamente en cero, así que hay que decidir cuántos dígitos de la fracción convertida se usan. Asimismo, recuerde que las multiplicaciones son por el número *r*. Por eso, para convertir una fracción decimal al octal, tenemos que multiplicar las fracciones por 8, como muestra el Ejemplo 1-6.

EJEMPLO 1-6 Conversión de fracciones decimales a octal

Convierta el número decimal 0.513 a una fracción octal de tres dígitos:

$0.513 \times 8 = \mathbf{4}.104$	**Entero = 4**	Dígito más significativo
$0.104 \times 8 = \mathbf{0}.832$	**= 0**	
$0.832 \times 8 = \mathbf{6}.656$	**= 6**	
$0.656 \times 8 = \mathbf{5}.248$	**= 5**	Dígito menos significativo

La respuesta, a tres figuras significativas, se obtiene de los dígitos enteros. Note que el último dígito entero, **5**, es usado para rondar en base 8 en el penúltimo dígito, **6**, para obtener

$$(0.513)_{10} = (0.\mathbf{407})_8.$$ ■

La conversión de números decimales con partes enteras y fraccionarias se realiza convirtiendo cada parte por separado y después combinando los dos resultados. Usando los resultados de los Ejemplos 1-3 y 1-6, obtenemos

$$(153.513)_{10} = (231.407)_8$$

1-4 CÓDIGOS DECIMALES

El sistema numérico binario es el sistema más natural para una computadora, pero las personas están acostumbradas al sistema decimal. Una posibilidad para resolver esa diferencia es convertir números decimales a binario, realizar todos los cálculos en binario y reconvertir los resultados binarios a decimal. Este método requiere que guardemos los números decimales en la computadora de una manera que facilite la conversión a binario. Como la computadora sólo acepta valores binarios, hay que representar los dígitos decimales mediante un código de 1 y 0. También es posible realizar las operaciones aritméticas directamente con números decimales cuando se guardan en la computadora de forma codificada.

Un *código binario* de n bits es un grupo de n bits que implica hasta 2^n combinaciones diferentes de 1 y 0, donde cada combinación representa un elemento del conjunto codificado. Un conjunto de cuatro elementos se puede codificar con un código binario de 2 bits, donde cada elemento está asignado a una de las siguientes combinaciones binarias: 00, 01, 10, 11. Un conjunto de 8 elementos requiere un código de 3 bits, y un conjunto de 16 elementos requiere un código de 4 bits. Las combinaciones binarias de un código de n bits se pueden determinar contando en binario de 0 a $2^n - 1$. A cada elemento hay que asignarle una única combinación binaria, no está permitido que dos elementos tengan el mismo valor; si no, el código es ambiguo.

Un código binario tendrá algunas combinaciones binarias sin asignar si el número de elementos en un conjunto no es una potencia de 2. Los 10 dígitos decimales forman un conjunto así. Un código binario que distingue entre 10 elementos tiene que contener al menos cuatro bits, pero 6 de las 16 combinaciones posibles permanecerán sin asignar. Se pueden obtener numerosos códigos binarios diferentes colocando los cuatro bits con 10 combinaciones diferentes. El código usado más común para los dígitos decimales es la asignación binaria directa enumerada en la Tabla 1-2. Se llama *decimal codificado en binario* (en inglés *binary-coded decimal*) o BCD. También son posibles otros códigos decimales, algunos de ellos se presentan en el Capítulo 3.

La Tabla 1-3 muestra un código de 4 bits para cada dígito decimal. Un número con n dígitos decimales requiere $4n$ bits en BCD. Así, el número decimal 396 se representa en BCD con 12 bits como

0011 1001 0110

□ **TABLA 1-3**
Decimal codificado en binario (BCD)

Símbolo decimal	Dígito BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

donde cada grupo de cuatro bits representa un dígito decimal. Un número decimal en BCD es lo mismo que su número equivalente binario si es un número entre 0 y 9, incluido. Un número BCD más grande que 10 tiene una representación diferente de su número binario equivalente, aunque ambos contienen 1 y 0. Además, las combinaciones binarias 1010 hasta 1111 no se usan y no tienen significado en el código BCD.

Considere decimal 185 y su valor correspondiente en BCD y binario:

$$(185)_{10} = (0001 \quad 1000 \quad 0101)_{BCD} = (10111001)_2$$

El valor BCD tiene 12 bits, pero su número equivalente binario necesita solamente 9 bits. Es obvio que un número en BCD necesita más bits que su valor binario equivalente. No obstante, hay una ventaja en el uso de números decimales porque los datos de entrada y salida se manejan por personas que usan el sistema decimal. Números BCD son números decimales y no binarios, aunque están representados con bits. La única diferencia entre un número decimal y BCD es que los números decimales se escriben con los símbolos 0, 1, 2, ..., 9, y los números BCD usan los códigos binarios 0000, 0001, 0010, ..., 1001.

Suma en BCD

Considere la suma de dos dígitos decimales en BCD, junto con un posible acarreo igual a 1, resultado de un par de dígitos anteriores menos significantivos. Como ningún dígito excede de 9, la suma no puede ser mayor que $9 + 9 + 1 = 19$, donde 1 es el acarreo. Supongamos que sumamos los dígitos BCD como si tuviéramos números binarios. Entonces, la suma binaria produce un resultado de 0 a 19. En binario, esto será de 0000 a 10011, pero en BCD, podría ser de 0000 a 1 1001, donde el primer 1 es el acarreo y los siguientes cuatro bits son la suma en BCD. Si la suma binaria es menor que 1010 (sin acarreo), el dígito BCD correspondiente es correcto. Pero si la suma binaria es mayor o igual que 1010, el resultado es un dígito BCD no válido. La adición de 6 en binario, $(0110)_2$, a la suma, lo convierte en el dígito correcto y además produce el acarreo decimal requerido. La razón es que la diferencia de un acarreo de la posición más significativa de la suma binaria y un acarreo decimal es $16 - 10 = 6$. Por eso, el acarreo decimal y el dígito correcto de la suma BCD se fuerzan sumando un 6 en binario. Considere el siguiente ejemplo de suma de tres dígitos en BCD.

EJEMPLO 1-7 Adición BCD

110	Acarreo BCD 1	**1**	**1**	
448		0100	0100	1000
+489		+0100	+1000	+1001
937	Suma binaria	1001	1101	**1** 0001
	Sumar 6		+0110	+0110
	BCD suma		**1** 0011	**1** 0111
	BCD resultado	1001	0011	0111

Para cada posición, se suman los dos dígitos BCD como si fuesen dos números binarios. Si la suma binaria es mayor que 1001, sumamos 0110 para obtener la suma de BCD correcta y un acarreo. En la columna derecha, la suma binaria es igual a 17. La presencia del acarreo indica

que la suma es mayor que 16 (indudablemente mayor que 9), con lo cual se necesita una corrección. La adición de 0110 produce la suma correcta en BCD, 0111 (7), y un acarreo de 1. En la siguiente columna, la suma binaria es 1101 (13), un dígito BCD no válido. La adición de 0110 produce la suma BCD correcta, 0011 (3), y un acarreo de 1. En la última columna, la suma es igual a 1001 (9) y es el dígito BCD correcto. ■

Bit de paridad

Para detectar errores en la comunicación y el procesamiento de datos, a veces se añade un bit adicional a una palabra de código binario para definir su paridad. Un bit de paridad es un extra bit incluido para conseguir que la cantidad de 1 en la palabra de código resultante sea o par o impar. Considere los dos caracteres siguientes y su paridad par o impar:

	Con paridad par	Con paridad impar
1000001	01000001	11000001
1010100	11010100	01010100

En cada caso usamos el bit extra en la posición más a la izquierda del código para producir un número par de 1 en el caracter para la paridad par o un número impar de 1 en el carácter para la paridad impar. En general, se usan ambas paridades, siendo la paridad par la más común. Se puede usar paridad tanto con numeros binarios como con códigos, incluyendo ASCII para los caracteres, y se puede emplazar el bit de paridad en cualquier posición fija del código.

El bit de paridad es útil para detectar errores durante la transmisión de información de un sitio a otro. Asumiendo que se usa paridad par, el caso más simple se trata de la manera siguiente: se genera un bit de paridad par (o impar) en el emisor para todos los caracteres ASCII de 7 bits; se transmiten los caracteres de 8 bits, que incluyen los bits de paridad, hacia su destino. En el destino se comprueba la paridad de cada carácter; si la paridad de un carácter recibido no es par (impar), significa que al menos un bit ha cambiado su valor durante la transmisión. Este método detecta uno, tres o cada número impar de errores en cada carácter transmitido. Un número par de errores no se detecta. Otros códigos de detección de errores, de los cuales algunos se basan en bits de paridad, pueden ser necesarios para vigilar los números pares de errores. Lo que se hace después de detectar un error depende de la aplicación particular. Una posibilidad es pedir una retransmisión del mensaje suponiendo que el error fue por azar y no ocurrirá otra vez. Por eso, el receptor, si detecta un error de paridad, devuelve un NAK (reconocimiento negativo, del inglés *negative acknowledge*), que es un carácter de control que consiste en 8 bits con paridad par, 10010101, de la Tabla 1-5. Si no se detecta ningún error, el receptor devuelve un carácter de control ACK (confirmación, del inglés *acknowledge*), 00000110. El emisor contestará a un NAK transmitiendo otra vez el mismo mensaje, hasta que se recibe la paridad correcta. Si la transmisión todavía tiene fallos después de un cierto número de intentos, se indica un mal funcionamiento en el camino de transmisión.

1-5 CÓDIGOS GRAY

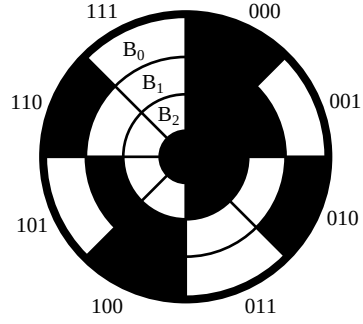
Cuando se cuenta adelante o atrás usando códigos binarios, el número de bits que cambian de un valor binario a otro varía. Esto se muestra en el código binario para los dígitos octales a la izquierda de la Tabla 1-4. Como contamos de 000 a 111 y «saltamos» a 000, el número de bits que cambia entre los valores binarios fluctúa de 1 a 3.

□ **TABLA 1-4**
Código Gray

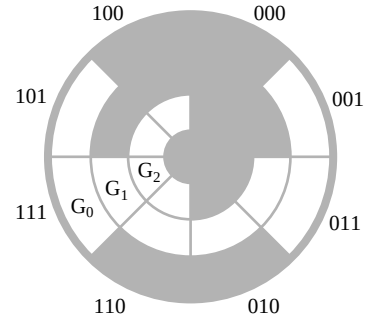
Código binario	Bits cambiados	Código Gray	Bits cambiados
000		000	
	1		1
001		001	
	2		1
010		011	
	1		1
011		010	
	3		1
100		110	
	1		1
101		111	
	2		1
110		101	
	1		1
111		100	
	3		1
000		000	

En muchas aplicaciones, múltiples cambios de bits como las cuentas circulares no presentan problemas. Pero hay aplicaciones donde un cambio de más de un bit contando hacia adelante o atrás puede causar problemas serios. Uno de estos problemas se ilustra mediante un codificador óptico de posición angular mostrado en la Figura 1-3(a). El codificador es un disco fijado en un eje giratorio para medir la posición rotatoria del eje. El disco contiene áreas transparentes para el 1 binario y opacas para el 0. Una fuente de luz está posicionada en un lado del disco, y los sensores ópticos, uno para cada de los bits a codificar, se encuentran en el otro lado del disco. Si hay una región transparente entre la fuente y el sensor, el sensor reacciona a la luz con una salida binaria igual a 1. Si hay una región opaca entre la fuente y el sensor, el sensor reacciona a la oscuridad con una salida binaria igual a 0.

Sin embargo, el eje giratorio puede estar en una posición angular. Por ejemplo, suponga que el eje y el disco se posicionan de manera que los sensores están justamente en el borde entre 011 y 100. En este caso, los sensores de las posiciones B_2, B_1 y B_0 tienen la luz parcialmente bloqueada. En una situación como esa no está claro si los sensores ven luz o oscuridad. Como resultado, cada sensor puede producir o 1 o 0. Así el número binario codificado resultante para un valor entre 3 y 4 puede ser 000, 001, 010, 011, 100, 101, 110 o 111. Tanto 011 como 100 serán adecuados en este caso, pero los otros 6 valores son claramente erróneos.



(a) Código binario para las posiciones de 0 a 7



(b) Código Gray para las posiciones de 0 a 7

□ **FIGURA 1-3**
Codificador óptico de posición angular

La solución a este problema parece evidente si uno se da cuenta de que en los casos donde cambia sólo un bit de valor al siguiente o anterior, este problema no puede ocurrir. Por ejemplo, si los sensores están en el borde entre 2 y 3, el código resultante es o 010 o 011, los dos son satisfactorios. Si cambiamos la codificación de los valores de 0 a 7 de la manera que solo cambia un bit contando hacia delante o atrás (incluyendo saltos de 7 a 0), entonces la codificación va a ser satisfactoria para todas las posiciones. Un código con la propiedad de que un solo bit cambia durante la cuenta es un Código Gray. Hay varios códigos Gray para cada conjunto de n enteros consecutivos, si n es par.

Un Código Gray para los dígitos octales, llamado *Código Gray binario reflectado* (del inglés *binary reflected Gray code*), aparece a la derecha de la Tabla 1-4. Note que el orden para contar códigos binarios es ahora 000, 001, 011, 010, 110, 111, 101, 100 y 000. Si queremos códigos binarios para su procesamiento, podemos construir un circuito binario o usar software que convierta estos códigos a binario antes de usarlos en el siguiente proceso de la información.

La Figura 1-3(b) presenta el codificador óptico de posición angular usando el Código Gray de la Tabla 1-4. Note que cada dos segmentos adyacentes en el disco sólo tienen una región transparente para uno y opaca para el otro. El Código Gray lleva su nombre por Frank Gray quien patentó su uso para codificadores ópticos de posición angular en 1953.

El codificador óptico de posición angular ilustra un uso del concepto del Código Gray. Hay muchos otros usos parecidos donde una variable física, como posición o voltaje, tiene un rango continuo de valores que se convierte a una representación digital. Un uso bastante diferente de códigos Gray aparece en circuitos lógicos CMOS (*Complementary Metal Oxide Semiconductor*) de bajo consumo que cuentan hacia delante y atrás. En CMOS, solamente se consume energía cuando cambia un bit. Para los códigos de ejemplo de la Tabla 1-4 con conteo continuo (o hacia delante o atrás), hay 14 cambios de bits contando en binario para cada 8 cambios de bits usando el Código Gray. Así, la energía consumida en las salidas del contador de Gray es solamente 57% de lo que se consume en las salidas del contador binario.

Un Código Gray para una secuencia de cuenta de n palabras de código binario (n tiene que ser par) puede ser construida sustituyendo cada de los primeros $n/2$ números de la secuencia por una palabra de código que consista en un 0 seguido de la paridad par para cada bit de la palabra de código binario con el bit a su izquierda. Por ejemplo, para la palabra de código binario 0100, la palabra de Código Gray es 0, paridad (0,1), paridad (1,0), paridad (0,0) = 0110. Después, coge la secuencia de números formada y copiala en orden inverso con el 0 más a la izquierda sustituido por 1. Esta nueva secuencia proporciona las palabras de Código Gray para los siguientes $n/2$ de las n palabras de código originales. Por ejemplo, para códigos BCD, las primeros cinco palabras son 0000, 0001, 0011, 0010 y 0110. Invirtiendo el orden de estos códigos y reemplazando el 0 más a la izquierda con 1, obtenemos 1110, 1010, 1011, 1001 y 1000 para los últimos cinco Códigos Gray.

Para casos especiales donde los códigos binarios originales son de 0 a $2^n - 1$, cada palabra de Código Gray puede formarse directamente de la palabra del código binario correspondiente copiando su bit más a la izquierda y después reemplazando cada uno de los bits sobrantes con el bit de paridad par para el bit del número y el bit a su izquierda.

1-6 CÓDIGOS ALFANUMÉRICOS

Muchas aplicaciones para computadoras digitales requieren el manejo de datos que no solamente consisten en números, sino también en letras. Por ejemplo, una compañía de seguros con miles de asegurados usa una computadora para procesar sus ficheros. Para representar los nom-

bres y otra información pertinente, es necesario formular un código binario para las letras del alfabeto. Además, el mismo código binario tiene que representar números y caracteres especiales como $. Cada conjunto alfanumérico en inglés es un conjunto de elementos que incluye los 10 dígitos decimales, las 26 letras del alfabeto, y algunos (más que tres) caracteres especiales. Si se incluyen solamente las mayúsculas, necesitamos un código binario de al menos seis bits, y si se incluyen las mayúsculas y las minúsculas necesitamos un código binario de al menos siete bits. Los códigos binarios tienen un papel muy importante en las computadoras digitales. Los códigos tienen que ser binarios, porque la computadora solamente entiende 1 y 0. Note que la codificación binaria cambia solamente los símbolos, no el significado de los elementos codificados.

Código ASCII para caracteres

El código estándar para caracteres alfanuméricos se llama ASCII (Código estandarizado americano para intercambio de información, *American Standard Code for Information Interchange*). Usa siete bits para codificar 128 caracteres, según se muestra en la Tabla 1-5. Los siete bits del código se indican como B_1 hasta B_7, donde B_7 es el bit más significativo. Note que los tres bits más significativos del código determinan la columna y los cuatro bits menos significantes la fila de la tabla. La letra A, por ejemplo, es representada en ASCII por 1000001 (columna 100, fila 0001). El código ASCII contiene 94 caracteres que pueden ser imprimidos y 34 caracteres no imprimibles usados para varias funciones de control. Los caracteres imprimibles consisten en 26 letras mayúsculas, 26 letras minúsculas, 10 cifras y 32 caracteres especiales imprimibles como %, @, y $.

Los 34 caracteres de control se indican en la tabla de ASCII con nombres abreviados. Se muestran otra vez debajo de la tabla con sus nombres completos funcionales. Los caracteres de control se usan para el encaminamiento de datos y colocar el texto impreso en un formato predefinido. Hay tres tipos de caracteres de control: efectos de formato, separadores de información, y caracteres de control de comunicación. Los efectos de formato son caracteres que controlan el diseño de la impresión. Incluyen los controles conocidos de la máquina de escribir como el retroceso (*backspace*, BS), tabulador horizontal (*horizontal tabulation*, HT), y retorno de carro (*carriage return*, CR). Los separadores de información se usan para separar los datos en divisiones, por ejemplo, párrafos y páginas. Incluyen caracteres como el separador de registro (*record separator*, RS) y separador de ficheros (*file separator*, FS). Los caracteres de control de comunicaciones se usan durante la transmisión de un texto de un sitio a otro. Ejemplos de caracteres de control de comunicación son STX (inicio de de texto, *start of text*) y ETX (final de texto, *end of text*), que se usan para enmarcar un mensaje de texto transmitido mediante comunicación sobre hilos.

ASCII es un código de 7 bits, pero la mayoría de las computadoras manipulan una cantidad de 8 bits como una unidad llamada *byte*. Por eso, se guardan los caracteres ASCII, normalmente, uno por byte, con el bit más significativo puesto en 0. El bit extra se usa a veces para fines específicos, dependiendo de la aplicación. Por ejemplo, algunas impresoras reconocen 128 caracteres adicionales de 8 bits, con el bit más significativo puesto en 1. Estos caracteres habilitan la impresora para producir símbolos adicionales, como por ejemplo los del alfabeto griego o caracteres con marcas de acentos como se usan en idiomas diferentes al inglés.

UNICODE Hay disponible un apéndice sobre Unicode, un código estándar de 16 bits para representar los símbolos y gráficos de lenguajes de todo el mundo, en la página Web de (http://www.librosite.net/mano) de este libro.

□ **TABLA 1-5**
American Standard Code for Information Interchange (ASCII)

	$B_7B_6B_5$							
$B_4B_3B_2B_1$	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Caracteres de control:

NULL	NULL	DLE	Data link escape
SOH	Inicio del cabecero	DC1	Control de dispositivo 1
STX	Inicio del texto	DC2	Control de dispositivo 2
ETX	Fin del texto	DC3	Control de dispositivo 3
EOT	Fin de la transmisión	DC4	Control de dispositivo 4
ENQ	Petición	NAK	Acknowledge negativo
ACK	Confirmación	SYN	Espera Síncrona
BEL	Timbre	ETB	Fin del bloque de transmisión
BS	Retroceso	CAN	Cancelar
HT	Tab. horizontal	EM	Fin del medio
LF	Line feed	SUB	Sustituir
VT	Tab. vertical	ESC	Escape
FF	Form feed	FS	Separador de fichero
CR	Retorno de carro	GS	Separador de grupo
SO	Desplazamiento hacia fuera	RS	Separador de registro
SI	Desplazamiento hacia dentro	US	Separador de unidad
SP	Espacio	DEL	Borrar

1-7 SUMARIO DEL CAPÍTULO

En este capítulo hemos presentado los sistemas digitales y las computadoras digitales y hemos ilustrado porque estos sistemas usan señales con dos valores solamente. Hemos introducido brevemente la estructura de la computadora mediante un diagrama de bloques discutiendo a la vez la naturaleza de los bloques. Se han presentado los conceptos de sistemas numéricos, incluyendo base y punto de base. Por su correspondencia con las señales de dos valores, se han presentado en detalle los números binarios. El sistema de numeración octal (base 8) y hexadecimal (base 16) también han sido enfatizados, al ser útiles como notación taquigráfica para el código binario. Las operaciones aritméticas en otras bases distintas de 10 y la conversión de números de una base a otra se han cubierto. Por el predominio del sistema de numeración decimal en el uso común, se ha tratado el código BCD. Se ha presentado el bit de paridad como técnica para detección de errores, y el código Gray, que es crítico para ciertas aplicaciones, se ha descrito. Finalmente, se ha presentado la representación de información en forma de caracteres en vez de números mediante el código ASCII para el alfabeto inglés.

En los capítulos siguientes, trataremos la representación de números con signo y números de punto flotante. También introduciremos códigos adicionales para dígitos decimales. Aunque estos temas se ajustan bien en los temas incluidos en este capítulo, son difíciles de justificar sin asociarles con el hardware usado para implementar las operaciones que se denotan. Así, retrasamos su presentación hasta que examinemos su hardware asociado.

REFERENCIAS

1. GRAY, F.: *Pulse Code Communication*. U. S. Patent 2 632 058, March 17, 1953.

2. MANO, M. M.: *Computer Engineering: Hardware Design.* Englewood Cliffs, NJ: Prentice Hall, 1988.

3. MANO, M. M.: *Digital Design*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

4. MANO, M. M.: *Computer System Architecture*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1993.

5. PATTERSON, D. A., and HENNESSY, J. L.: *Computer Organization and Design: The Hardware/Software Interface.* 2nd ed. San Mateo, CA: Morgan Kaufmann, 1998.

6. TANENBAUM, A. S.: *Structured Computer Organization*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1999.

7. WHITE, R.: *How Computers Work*, Emeryville, CA: Ziff-Davis Press, 1993.

8. WILLIAMS, M. R.: *A History of Computing Technology*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

PROBLEMAS

El signo (+) indica problemas más avanzados y el asterisco (*) indica que hay una solución disponible en la dirección de Internet: http://www.librosite.net/Mano.

1-1. *Enumere los números binarios, octales, y hexadecimales de 16 a 31.

1-2. ¿Cuál es el número exacto de bits en una memoria que contiene **(a)** 48 Kbits; **(b)** 384 Mbits; **(c)** 8 Gbits?

1-3. ¿Cuál es el número decimal equivalente del entero binario más grande que se puede obtener con **(a)** 12 bits y **(b)** 24 bits?

1-4. *Convierta los números binarios siguientes a decimal: 1001101, 1010011.101 y 10101110.1001.

1-5. Convierta los siguientes números decimales a binario: 125, 610, 2003 y 18944.

1-6. Cada uno de los siguientes cinco números tiene una base diferente: $(11100111)_2$, $(22120)_3$, $(3113)_4$, $(4110)_5$, y $(343)_8$. ¿Cuál de los cinco números tienen el mismo valor en decimal?

1-7. *Convierta los números siguientes de una base dada a las otras tres bases enumeradas en la tabla:

Decimal	Binario	Octal	Hexadecimal
369.3125	?	?	?
?	10111101.101	?	?
?	?	326.5	?
?	?	?	F3C7.A

1-8. *Convierta los siguientes números decimales a las bases indicadas usando los métodos de los Ejemplos 1-3 y 1-6:

(a) 7562.45 a octal **(b)** 1938.257 a hexadecimal **(c)** 175.175 a binario

1-9. *Realice la siguiente conversión usando base 2 en vez de base 10 como base intermedia para la conversión:

(a) $(673.6)_8$ a hexadecimal **(b)** $(E7C.B)_{16}$ a octal **(c)** $(310.2)_4$ a octal

1-10. Realice las multiplicaciones binarias siguientes:

(a) 1101×1001 **(b)** 0101×1011 **(c)** 100101×0110110

1-11. +La división está compuesta por multiplicaciones y substracciones. Realice la división binaria $1011110 \div 101$ para obtener el cociente y el resto.

1-12. Hay una evidencia considerable en suponer que la base 20 ha sido usada históricamente para sistemas numéricos en algunas culturas.

(a) Escriba los dígitos para un sistema en base a 20, usando una extensión del mismo esquema de representación de dígitos empleado para hexadecimal.
(b) Convierta $(2003)_{10}$ a la base 20. **(c)** Convierta $(BCH.G)_{20}$ al decimal.

1-13. *En cada uno de los siguientes casos, determine el la base r:

(a) $(BEE)_r = (2699)_{10}$ **(b)** $(365)_r = (194)_{10}$

1-14. El cálculo siguiente ha sido realizado por una especie particular de pollos extraordinariamente inteligentes. Si la base r usada por el pollo corresponde a su numero total de dedos, cuantos dedos tiene el pollo en cada pata?

$$((35)_r + (24)_r) \times (21)_r = (1501)_r$$

1-15. *Represente los números decimales 694 y 835 en BCD, e indique después los pasos necesarios para formar su suma.

1-16. *Encuentre las representaciones binarias para cada uno de los siguientes números BCD:

(a) 0100 1000 0110 0111 **(b)** 0011 0111 1000.0111 0101

1-17. Enumere los números binarios equivalentes de 5 bits para 16 hasta 31 con un bit de paridad añadido en la posición más a la derecha dando paridad impar para la totalidad de números de 6 bits. Repita para paridad par.

1-18. Usando el procedimiento dado en la Sección 1-5, encuentre el Código Gray para dígitos hexadecimales.

1-19. +¿Cuál es el porcentaje de energía consumida por un contador continuo (o hacia delante o hacia detrás, no ambos) en las salidas de un contador de Código Gray binario comparado con un contador binario en función del número de bits, *n*, en los dos contadores?

1-20. ¿Qué posición del bit en código ASCII tiene que ser complementado para cambiar la letra ASCII representada de mayúsculas a minúsculas y al revés?

1-21. Escriba su nombre completo en ASCII, usando un código de 8 bits **(a)** con el bit más a la izquierda siempre en 0 y **(b)** con el bit más a la izquierda seleccionado para producir paridad par. Incluya un espacio entre los nombres y un punto después de él.

1-22. Decodifique el código ASCII siguiente: 1001010 1101111 1101000 1101110 0100000 1000100 1101111 1100101.

1-23. *Indique la configuración de bits que representa al número decimal 365 en **(a)** binario, **(b)** BCD, **(c)** ASCII.

1-24. Una computadora representa información en grupos de 32 bits. Cuantos enteros diferentes se pueden representar en **(a)** binario, **(b)** BCD, y **(c)** 8-bit ASCII, todos usando 32 bits?

CAPÍTULO

2

CIRCUITOS LÓGICOS COMBINACIONALES

En este capítulo estudiaremos las puertas, los elementos lógicos más sencillos usados en los sistemas digitales. Además, aprenderemos las técnicas matemáticas usadas en el diseño de circuitos con esas puertas y cómo diseñar circuitos eficientes en coste. Estas técnicas, que son fundamentales para diseñar casi todos los circuitos digitales, se basan en el Álgebra de Boole. Un aspecto del diseño es evitar circuitos innecesarios y costes excesivos, una meta que se cumple mediante una técnica llamada optimización. Los Mapas de Karnaugh proporcionan un método gráfico para mejorar el entendimiento de optimización y solucionar pequeños problemas de circuitos con «dos niveles». Se introducen los métodos más generales de optimización para circuitos con más de dos niveles. Se discuten los tipos de puertas lógicas características en la realización de los circuitos integrados actuales. Se presentan las puertas OR y NOR exclusiva, junto con las técnicas algebraicas asociadas.

En términos del diagrama del principio del Capítulo 1, los conceptos de este capítulo se pueden aplicar a la mayor parte de la computadora genérica. Las excepciones a esto son circuitos que son, principalmente, memorias, como cachés y RAM, y circuitos analógicos en el monitor y el controlador del disco duro. Sin embargo, con su uso por todas partes del diseño de la mayor parte de la computadora, lo que estudiaremos en este capítulo es fundamental para un entendimiento profundo de las computadoras y los sistemas digitales, y cómo están diseñados.

2-1 LÓGICA BINARIA Y PUERTAS

Los circuitos digitales son componentes de hardware que manipulan información binaria. Los circuitos se realizan con transistores e interconexiones en complejos dispositivos de semiconductores llamados *circuitos integrados*. A cada circuito básico se le denomina *puerta lógica*. Por simplicidad en el diseño, modelamos los circuitos electrónicos basados en transistores como puertas lógicas. Así, el diseñador no tiene que preocuparse por la electrónica interna de cada una de las individuales, sino solamente por sus propiedades lógicas externas. Cada puerta realiza una operación lógica específica. Las salidas de las puertas se aplican a las entradas de otras puertas para formar un circuito digital.

Para describir las propiedades operacionales de los circuitos digitales es necesario introducir una notación matemática que especifica la operación de cada puerta y que puede ser usada para analizar y diseñar circuitos. Este sistema de lógica binaria es una clase de sistema matemático que se denomina *Álgebra de Boole*. El nombre es en honor al matemático inglés George Boole, quien publicó un libro en 1854 introduciendo la teoría matemática de la lógica. El álgebra específica de Boole que estudiaremos se usa para describir la interconexión de las puertas digitales y para diseñar circuitos lógicos a través del uso de expresiones booleanas. Primero introducimos el concepto de lógica binaria e indicamos su relación con las puertas digitales y las señales binarias. Después presentamos las propiedades del Álgebra de Boole, junto con otros conceptos y métodos útiles en el diseño de circuitos lógicos.

Lógica binaria

La lógica binaria trabaja con variables binarias, que pueden tomar dos valores discretos, y con las operaciones lógicas matemáticas aplicadas a esas variables. A los dos valores que pueden tomar las variables se les pueden llamar por diferentes nombres, como se mencionó en la Sección 1-1, pero para nuestro propósito, es conveniente pensar en términos de valores binarios y asignar 1 o 0 a cada variable. En la primera parte de este libro, se designan a las variables con las letras del alfabeto, como *A*, *B*, *C*, *X*, *Y*, y *Z*. Más tarde se extiende esta notación para incluir cadenas de letras, números y caracteres especiales. Asociados con las variables binarias hay tres operaciones lógicas llamadas AND, OR y NOT:

1. AND. Esta operación está representada por un punto o por la ausencia de un operador. Por ejemplo, $Z = X \cdot Y$ o $Z = XY$ se lee «*Z* es igual a *X* AND *Y*». La operación lógica AND se interpreta de manera que $Z = 1$ si y solamente si $X = 1$ e $Y = 1$; de lo contrario es $Z = 0$. (Recuerde que *X*, *Y* y *Z* son variables binarias y solamente pueden tener los valores 1 o 0.)
2. OR. Esta operación se representa por el símbolo «más». Por ejemplo, $Z = X + Y$ se lee «*Z* es igual a *X* OR *Y*», lo que significa que $Z = 1$ si $X = 1$ o si $Y = 1$, o si los dos $X = 1$ e $Y = 1$. $Z = 0$ si y solamente si $X = 0$ e $Y = 0$.
3. NOT. Esta operación está representada por una barra encima de la variable. Por ejemplo, $Z = \overline{X}$ se lee «*Z* es igual a NOT *X*,» lo que significa que *Z* es lo que *X* no es. En otras palabras, si $X = 1$, entonces $Z = 0$, pero si $X = 0$, entonces $Z = 1$. A la operación NOT se le denomina también como operación complementaria, ya que cambia un 1 a 0 y un 0 a 1.

Lógica binaria se parece a la aritmética binaria, y las operaciones AND y OR se parecen a la multiplicación y la suma, respectivamente. Por eso los símbolos usados para la AND y la OR

son los mismos que los que se usan para la multiplicación y la suma. Sin embargo, no se debe confundir la lógica binaria con la aritmética binaria. Uno debería darse cuenta de que una variable aritmética define a un número que se puede componer de muchos dígitos, mientras una variable lógica siempre es 1 o 0. Las siguientes ecuaciones definen la operación lógica OR:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

Esto se parece a la suma binaria, excepto para la última operación. En la lógica binaria, tenemos que $1 + 1 = 1$ (léase «uno o uno es igual a uno»), pero en la aritmética binaria tenemos $1 + 1 = 10$ (léase «uno más uno es igual a dos»). Para evitar ambigüedad, a veces se usa el símbolo $\vee$ para la operación OR en vez del símbolo $+$. Pero mientras no se mezclen operaciones aritméticas y lógicas, en cada parte se puede usar el símbolo $+$ con su propio significado independiente.

Las siguientes ecuaciones definen la operación lógica AND:

$$0 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 0 = 0$$

$$1 \cdot 1 = 1$$

Esta operación es idéntica a la multiplicación binaria, con tal de que se use solamente un único bit. Los símbolos alternativos para el $\cdot$ de la AND y el $+$ de la OR, son los símbolos $\wedge$ y $\vee$, respectivamente, que representan operaciones conjuntivas y disyuntivas en cálculos proposicionales.

Para cada combinación de los valores de variables binarias como *X* e *Y*, hay un valor de *Z* especificado por la definición de la operación lógica. Las definiciones pueden ser enumeradas de forma compacta en una tabla de verdad. Una *tabla de verdad* para una operación es una tabla de combinaciones de las variables binarias que muestran la relación entre los valores que toman las variables y los valores del resultado de la operación. Las tablas de verdad para las operaciones AND, OR y NOT se muestran en la Tabla 2-1. Las tablas enumeran todas las combinaciones posibles de valores para dos variables y el resultado de la operación. Demuestran claramente la definición de las tres operaciones.

Puertas lógicas

Las puertas lógicas son circuitos electrónicos que operan con una o más señales de entrada para producir una señal de salida. Las señales eléctricas como voltaje o corriente existen en todas las partes de un sistema digital en cada uno de los dos valores definidos. Los circuitos que operan con voltajes responden a dos rangos separados de voltajes que representan una variable binaria igual a un 1 lógico o a un 0 lógico, como se ilustra en la Figura 1-1. Los terminales de entrada

□ **TABLA 2-1**
Tablas de verdad para las tres operaciones lógicas básicas

AND

X	Y	$Z = X \cdot Y$
0	0	0
0	1	0
1	0	0
1	1	1

OR

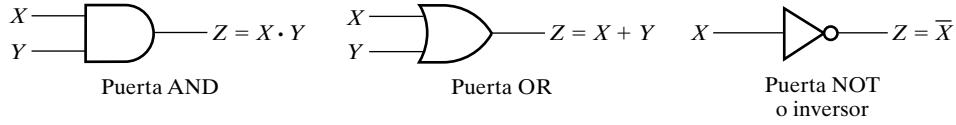
X	Y	$Z = X + Y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

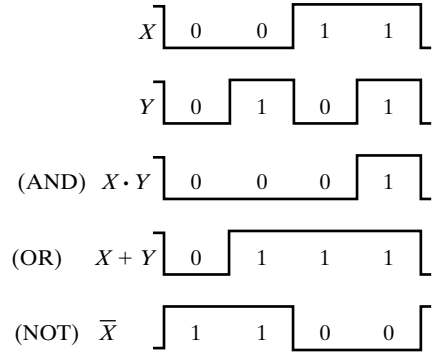
X	$Z = \overline{X}$
0	1
1	0

de las puertas lógicas aceptan señales binarias dentro del rango permitido y responden a los terminales de salida con señales binarias que caen dentro de un rango específico. Las regiones intermedias entre los rangos permitidos de la figura se cruzan solamente durante los cambios de 1 a 0 o de 0 a 1. A estos cambios se le llaman transiciones, y las regiones intermediarias se llaman regiones de tránsito.

Los símbolos gráficos usados para designar los tres tipos de puertas —AND, OR y NOT— se muestran en la Figura 2-1(a). Las puertas son circuitos electrónicos que producen los equivalentes a las señales de salida de 1 lógico y 0 lógico, de acuerdo con sus respectivas tablas de verdad, si se aplican el equivalente de las señales de entrada de 1 lógico y 0 lógico. Las dos señales de entrada X e Y de las puertas AND y OR toman una de cuatro combinaciones: 00, 01, 10, o 11. Estas señales de entrada se muestran en los diagramas de tiempos de la Figura 2-1(b), junto con los diagramas de tiempos de las señales de salida correspondientes a cada tipo de puerta. El eje horizontal de un diagrama de tiempos representa el tiempo, y el eje vertical muestra una señal cuando cambia entre los dos posibles niveles de voltaje. El nivel bajo representa

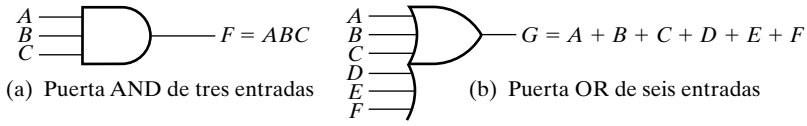


(a) Símbolo gráfico



(b) Diagrama de tiempos

□ **FIGURA 2-1**
Puertas lógicas digitales



(a) Puerta AND de tres entradas (b) Puerta OR de seis entradas

□ **FIGURA 2-2**
Puertas con más que dos entradas

el 0 lógico y el nivel alto representa el 1 lógico. La puerta AND se corresponde con una señal de salida a 1 lógico cuando las dos señales de entrada son un 1 lógico. La puerta OR responde con una señal de salida a 1 lógico si una de las señales de entrada es un 1 lógico. A la puerta NOT se le llama frecuentemente como *inversor*. La razón para este nombre es evidente por su respuesta en el diagrama de tiempos. La señal lógica de la salida es una versión invertida de la señal lógica de la entrada *X*.

Las puertas AND y OR pueden tener más de dos entradas. En la Figura 2-2 se muestra una puerta AND con tres entradas y una puerta OR con seis entradas. La puerta AND de tres entradas responde con una salida a 1 lógico si las tres entradas son 1 lógico. La salida es un 0 lógico si alguna de las entradas es un 0 lógico. La puerta OR de seis entradas responde con un lógico 1 si alguna de las entradas es un 1 lógico; su salida será un 0 lógico solamente cuando todas las entradas son 0 lógico.

2-2 ÁLGEBRA DE BOOLE

El Álgebra de Boole que presentamos es un álgebra que trata con variables binarias y operaciones lógicas. Las variables se indican con las letras del alfabeto y las operaciones básicas son AND, OR y NOT (complemento). Una expresión *booleana* es una expresión algebraica formada por variables binarias, las constantes 0 y 1, los símbolos de operación lógicos y paréntesis. Una función booleana se puede describir con una ecuación booleana que se compone de una variable binaria que identifica la función seguida por un símbolo de igualdad y una expresión booleana. Opcionalmente, al identificador le pueden seguir paréntesis que rodean a una lista de las variables de la función separadas por comas. Una *función booleana con única salida* se tabula a partir de cada combinación posible de valores 0 y 1 entre las variables de la función al valor 0 o 1. Una *función booleana con salida múltiple* se tabula a partir de cada combinación posible de valores 0 y 1 entre las variables de la función a combinaciones de 0 y 1 entre las salidas de la función. Considere un ejemplo de una ecuación booleana que representa a la función *F*:

$$F(X, Y, Z) = X + \bar{Y}Z$$

A las dos partes de la expresión, *X* y $\bar{Y}Z$, se le llaman *términos* de la expresión de *F*. La función *F* es igual a 1 si el término *X* es igual a 1 o si el término $\bar{Y}Z$ es igual a 1 (es decir, ambos $\bar{Y}$ y *Z* son iguales a 1). De otro modo, *F* es igual a 0. La operación complemento determina que si $\bar{Y} = 1$, *Y* tiene que ser igual a 0. Por tanto, podemos decir que $F = 1$ si $X = 1$ o si $Y = 0$ y $Z = 1$. Una ecuación booleana expresa la relación lógica entre variables binarias. Se evalúa determinando el valor binario de la expresión para todas las combinaciones posibles de valores para las variables.

Se puede representar una función booleana con una tabla de verdad. Una *tabla de verdad* para una función es una lista de todas las combinaciones de 1 y 0 que se pueden asignar a las variables binarias y una lista que indica el valor de la función para cada combinación binaria.

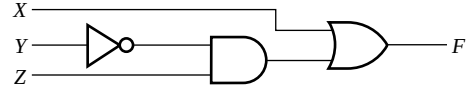
□ **TABLA 2-2**
Tabla de verdad de la función $F = X + \bar{Y}Z$

X	*Y*	*Z*	*F*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Las tablas de verdad para las operaciones lógicas de la Tabla 2-1 son casos especiales de tablas de verdad para funciones. El número de filas en una tabla de verdad es 2^n, donde n es el número de variables de la función. Las combinaciones binarias para la tabla de verdad son los números binarios de n-bit que corresponden a la cuenta en decimal de 0 a $2^n - 1$. La Tabla 2-2 muestra la tabla de verdad de la función $F = X + \bar{Y}Z$. Hay ocho posibles combinaciones binarias que asignan bits a las tres variables *X*, *Y* y *Z*. La columna etiquetada como *F* contiene 0 o 1 para cada una de las combinaciones. La tabla indica que la función es igual a 1 si $X = 1$ y si $Y = 0$ y $Z = 1$. De otro modo, la función es igual a 0.

Una expresión algebraica para una función booleana puede transformarse en un diagrama de un circuito compuesto por puertas lógicas que efectúan la función. El diagrama lógico del circuito para la función *F* se muestra en la Figura 2-3. Un inversor en la entrada *Y* genera el complemento, $\bar{Y}$. Una puerta AND opera con $\bar{Y}$ y *Z*, y una puerta OR combina *X* y $\bar{Y}Z$. En los diagramas lógicos de los circuitos, las variables de la función *F* se toman como entradas del circuito, y la variable binaria *F* se toma como salida del circuito. Si el circuito tiene una única salida, *F* es una función de salida única. Si el circuito tiene múltiples salidas, la función *F* es una función de salida múltiple que requiere de múltiples ecuaciones para representar sus salidas. Las puertas del circuito están interconectadas por hilos que llevan las señales lógicas. A los circuitos lógicos de este tipo se les llama *circuitos lógicos combinacionales*, porque las variables están «combinadas» por las operaciones lógicas. Esto es lo contrario a la lógica secuencial, que se trata en el Capítulo 6, donde se almacenan y combinan las variables en función del tiempo.

Una función booleana se puede representar en una tabla de verdad de una sola manera. No obstante, si la función tiene forma de ecuación algebraica, puede ser expresada de diferentes maneras. La expresión particular usada para representar la función determina la interconexión de las puertas en el diagrama lógico del circuito. Manipulando una expresión booleana conforme con reglas algebraicas booleanas, muchas veces es posible obtener una expresión más simple para la misma función. Ésta función más simple reduce el número de puertas en el circuito y el número de entradas de las puertas. Para ver como se logra esto, es necesario estudiar primero las reglas básicas del Álgebra de Boole.



□ **FIGURA 2-3**
Diagrama lógico de circuito para $F = X + \bar{Y}Z$

Identidades básicas del Álgebra de Boole

En la Tabla 2-3 se enumeran las identidades básicas del Álgebra de Boole. La notación está simplificada omitiendo el símbolo de la AND siempre que no lleve a ninguna confusión. Las nueve primeras identidades indican la relación entre una única variable X, su complemento $\overline{X}$, y las constantes binarias 0 y 1. Las siguientes cinco identidades, de la 10 a la 14, son las mismas que en el álgebra ordinaria. Las tres últimas, de la 15 a la 17, no se usan en el álgebra ordinaria, pero son útiles para manipular expresiones booleanas.

Las reglas básicas enumeradas en la tabla han sido colocadas en dos columnas que demuestran la propiedad dual del Álgebra de Boole. El dual de una expresión algebraica se obtiene intercambiando las operaciones OR y AND y reemplazando los 1 por 0 y los 0 por 1. Una ecuación en una columna de la tabla se puede obtener de la ecuación correspondiente de la otra columna usando el dual de las expresiones en ambos lados del símbolo de igualdad. Por ejemplo, la relación 2 es la dual de la relación 1 porque la OR ha sido reemplazada por la AND y el 0 por el 1. Es importante darse cuenta de que, en la mayoría de las veces, el dual de la expresión no es igual que la expresión original, de manera que una expresión normalmente no puede ser reemplazada por su dual.

Las nueve identidades que involucran a una única variable se pueden verificar sencillamente sustituyendo cada uno de los posibles valores de X. Por ejemplo, para mostrar que $X + 0 = X$, sea $X = 0$ para obtener $0 + 0 = 0$, y después sea $X = 1$ para obtener $1 + 0 = 1$. Ambas ecuaciones son verdad conforme a la definición de la operación lógica OR. Cada expresión puede ser sustituida por la variable X en todas las ecuaciones booleanas enumeradas en la tabla. Así, en la identidad 3 y con $X = AB + C$, obtenemos

$$AB + C + 1 = 1$$

Véase que la identidad 9 expresa que la doble complementación restaura el valor original de la variable. Así, si X = 0, entonces $\overline{X} = 1$ y $\overline{\overline{X}} = 0 = X$.

Las identidades 10 y 11, las leyes conmutativas, expresan que el orden en que se escriben las variables no afecta el resultado usando las operaciones OR y AND. Las identidades 12 y 13, las leyes asociativas, expresan que el resultado aplicando una operación sobre tres variables es

□ **TABLA 2-3**
Identidades básicas del Álgebra de Boole

1.	$X = 0 = X$	2.	$X \cdot 1\ X$	
3.	$X + 1 = 1$	4.	$X \cdot 0 = 0$	
5.	$X + X = X$	6.	$X \cdot X = X$	
7.	$X + \overline{X} = 1$	8.	$X \cdot \overline{X} = 0$	
9.	$\overline{\overline{X}} = X$			
10.	$X + Y = Y + X$	11.	$XY = YX$	Conmutativa
12.	$X + (Y + Z) = (X + Y) + Z$	13.	$X(YZ) = (XY)Z$	Asociativa
14.	$X(Y + Z) = XY + XZ$	15.	$X + YZ = (X + Y)(X + Z)$	Distributiva
16.	$\overline{X + Y} = \overline{X} \cdot \overline{Y}$	17.	$\overline{X \cdot Y} = \overline{X} + \overline{Y}$	De DeMorgan

independiente del orden en que se apliquen, y asimismo, los paréntesis se puede quitar como en el siguiente caso:

$$X + (Y + Z) = (X + Y) + Z = X + Y + Z$$

$$X(YZ) = (XY)Z = XYZ$$

Estas dos leyes y la primera ley distributiva, Identidad 14, son bien conocidas del álgebra ordinaria, por eso no deberían causar ninguna dificultad. La segunda ley distributiva, dada por Identidad 15, es el dual de la ley ordinaria distributiva y no se basa en el álgebra ordinaria. Como se ilustró anteriormente, se puede reemplazar cada variable de una identidad por una expresión booleana, y la identidad es todavía válida. Así, considere la expresión $(A + B)(A + CD)$. Poniendo $X = A$, $Y = B$ y $Z = CD$, y aplicando la segunda ley distributiva, obtenemos

$$(A + B)(A + CD) = A + BCD$$

A las dos últimas identidades de la Tabla 2-3,

$$\overline{X + Y} = \bar{X} \cdot \bar{Y} \text{ y } \overline{X \cdot Y} = \bar{X} + \bar{Y}$$

se les denomina como Teorema de DeMorgan. Es un teorema muy importante y se usa para obtener el complemento de una expresión y de la función correspondiente. El Teorema de DeMorgan se puede ilustrar mediante tablas de verdad que asignan todos los valores binarios posibles a X e Y. La Tabla 2-4 muestra dos tablas de verdad que verifican la primera parte del Teorema de DeMorgan. En A, evaluamos $\overline{X + Y}$ para todos los valores posibles de X e Y. Esto se hace primero evaluando $X + Y$ y después calculando el complemento. En B, evaluamos $\bar{X}$ e $\bar{Y}$ y después las conectamos con una operación AND. El resultado es el mismo para las cuatro combinaciones binarias de X e Y, que verifican la identidad de la ecuación.

Véase el orden en que se realizan las operaciones al evaluar una expresión. En la parte B de la tabla, se evalúan primero los complementos de las variables particulares, seguida de la operación AND, justo como en el álgebra ordinaria con la multiplicación y la suma. En la parte A, se evalúa primero la operación OR. Después, notando que el complemento de una expresión como $X + Y$ se considera como NOT $(X + Y)$, evaluamos la expresión de dentro de los paréntesis y tomamos el complemento del resultado. Es usual excluir los paréntesis calculando el complemento de una expresión, y a la barra encima de una expresión entera la une. Así, $(X + Y)$ se expresa como $\overline{X + Y}$ cuando se indica el complemento de $X + Y$.

El Teorema de DeMorgan puede ser extendido a tres o más variables. El Teorema General de DeMorgan se puede expresar como

$$\overline{X_1 + X_2 + ... + X_n} = \bar{X}_1\bar{X}_2...\bar{X}_n$$

$$\overline{X_1X_2...X_n} = \bar{X}_1 + \bar{X}_2 + ... + \bar{X}_n$$

□ **TABLA 2-4**
Tablas de verdad para verificar el teorema de DeMorgan

A) X	Y	$X + Y$	$\overline{X + Y}$	**B)** X	Y	$\bar{X}$	$\bar{Y}$	$\bar{X} \cdot \bar{Y}$
0	0	0	1	0	0	1	1	1
0	1	1	0	0	1	1	0	0
1	0	1	0	1	0	0	1	0
1	1	1	0	1	1	0	0	0

Observe que la operación lógica cambia de OR a AND o de AND a OR. Además, se elimina el complemento de la expresión entera y se coloca encima de cada variable. Por ejemplo,

$$\overline{A + B + C + D} = \bar{A}\bar{B}\bar{C}\bar{D}$$

Manipulación algebraica

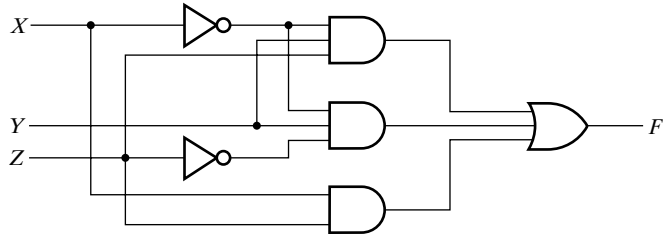
El Álgebra de Boole es un instrumento muy útil para simplificar circuitos digitales. Considere, por ejemplo, la función booleana representada por

$$F = \bar{X}YZ + \bar{X}Y\bar{Z} + XZ$$

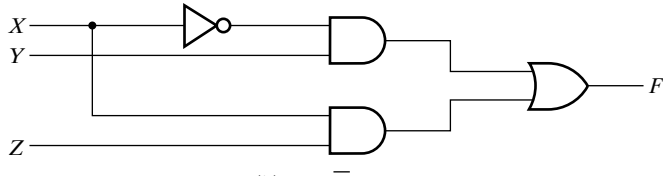
La implementación de ésta ecuación con puertas lógicas se muestra en la Figura 2-4(a). A las variables de entrada X y Z se le ha realizado el complemento con inversores para obtener $\bar{X}$ y $\bar{Z}$. Los tres términos de la expresión se realizan con tres puertas AND. La puerta OR forma la OR lógica de tres de los términos. Considere ahora una simplificación de la expresión para F aplicando algunas de las identidades listadas en la Tabla 2-3:

$$\begin{aligned} F &= \bar{X}YZ + \bar{X}Y\bar{Z} + XZ \\ &= \bar{X}Y(Z + \bar{Z}) + XZ \quad \text{con la identidad 14} \\ &= \bar{X}Y \cdot 1 + XZ \quad \text{con la identidad 7} \\ &= \bar{X}Y + XZ \quad \text{con la identidad 2} \end{aligned}$$

La expresión se reduce a sólo dos términos y puede ser realizada con puertas según se muestra en la Figura 2-4(b). Es obvio que el circuito de (b) es más simple que el de (a), ahora, ambos realizan la misma función. Es posible usar una tabla de verdad para verificar que dos implementaciones son equivalentes. Ésto se muestra en la Tabla 2-5. Como se expresa en la Figura 2-4(a), la función es igual a 1 si $X = 0$, $Y = 1$, y $Z = 1$; si $X = 0$, $Y = 1$, y $Z = 0$; o si X y Z son ambas 1. Esto produce los cuatro 1 para F en la parte (a) de la tabla. Como se expresa en la Figura 2-4(b), la función es igual a 1 si $X = 0$ e $Y = 1$ o si $X = 1$ y $Z = 1$. Esto produce los mismos



(a) $F = \bar{X}YZ + \bar{X}Y\bar{Z} + XZ$



(b) $F = \bar{X}Y + XZ$

□ **FIGURA 2-4**
Implementación de funciones booleanas con puertas

□ **TABLA 2-5**
Tabla de verdad para la función booleana

X	*Y*	*Z*	(a) *F*	(b) *F*
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

cuatro 1 en la parte (b) de la tabla. Como ambas expresiones producen las mismas tablas de verdad, se dice que son equivalentes. Por eso, los dos circuitos tienen las mismas salidas para todas las combinaciones binarias posibles de las tres variables de entrada. Cada circuito realiza la misma función, pero se prefiere la de menor número de puertas porque requiere menos componentes.

Si se implementa una ecuación booleana con puertas lógicas, cada término requiere una puerta, y cada variable dentro del término indica una entrada para la puerta. Definimos un *literal* como una variable única dentro de un término que puede estar complementado o no. La expresión para la función de la Figura 2-4(a) tiene tres términos y ocho literales; la de la Figura 2-4(b) tiene dos términos y cuatro literales. Reduciendo el número de términos, el número de literales, o ambos en una expresión booleana, muchas veces es posible obtener un circuito más sencillo. Se aplica el Álgebra de Boole para reducir una expresión con el fin de obtener un circuito más sencillo. Para funciones muy complejas es muy difícil encontrar la mejor expresión basada en sumas de términos y literales, aunque se usen programas de computadora. Ciertos métodos, sin embargo, para reducir expresiones, se incluyen frecuentemente en las herramientas por computadora para sintetizar circuitos lógicos. Estos métodos pueden obtener buenas soluciones, si no las mejores. El único método manual para el caso general es el procedimiento de intentar y probar a emplear las relaciones básicas y otras manipulaciones que uno va conociendo bien con el uso. Los siguientes ejemplos usan las identidades de la Tabla 2-3 para ilustrar algunas posibilidades:

1. $X + XY = X(1 + Y) = X$
2. $XY + X\bar{Y} = X(Y + \bar{Y}) = X$
3. $X + \bar{X}Y = (X + \bar{X})(X + Y) = X + Y$

Véase que el paso intermedio $X = X \cdot 1$ se ha omitido cuando se saca el factor X en la ecuación 1. La relación $1 + Y = 1$ es útil para eliminar términos redundantes, como se hace con el término XY en esta misma ecuación. La relación $Y + \bar{Y} = 1$ es útil para combinar dos términos, como se hace en la ecuación 2. Los dos términos combinados tienen que ser idénticos excepto en una variable, y esa variable tiene que estar complementada en un término y no complementada en el otro. La ecuación 3 está simplificada mediante la segunda ley distributiva (identidad 15 en la Tabla 2-3). A continuación hay tres ejemplos para simplificar expresiones booleanas:

4. $X(X + Y) = X + XY = X$
5. $(X + Y)(X + \bar{Y}) = X + Y\bar{Y} = X$
6. $X(\bar{X} + Y) = X\bar{X} + XY = XY$

Véase que los pasos intermediarios $XX = X = X \cdot 1$ han sido omitidos durante la manipulación de la ecuación 4. La expresión de la ecuación 5 está simplificada mediante la segunda ley distributiva. Aquí omitimos otra vez los pasos intermediarios $Y\bar{Y} = 0$ y $X + 0 = X$.

Las ecuaciones 4 a 6 son las duales de las ecuaciones 1 a 3. Recuerde que el dual de una expresión se obtiene cambiando AND por OR y OR por AND en todas partes (y 1 por 0 y 0 por 1 si aparecen en la expresión). El *principio de dualidad* del Álgebra de Boole expresa que una ecuación booleana permanece válida si tomamos el dual de la expresión en ambos lados del signo de igualdad. Por eso, las ecuaciones 4, 5 y 6 se pueden obtener tomando el dual de las ecuaciones 1, 2 y 3, respectivamente.

Junto con los resultados dados en las ecuaciones 1 a 6, el teorema siguiente, *teorema de consenso*, es útil a la hora de simplificar expresiones booleanas:

$$XY + \bar{X}Z + YZ = XY + \bar{X}Z$$

El teorema muestra que el tercer término, YZ, es redundante y se puede eliminar. Note que se asocian Y y Z con X y $\bar{X}$ en los primeros dos términos y que aparecen juntos en el término eliminado. La prueba del teorema de consenso se obtiene por la conexión AND entre YZ y $(X + \bar{X}) = 1$ y siguiendo después como se indica a continuación:

$$\begin{aligned} XY + \bar{X}Z + YZ &= XY + \bar{X}Z + YZ(X + \bar{X}) \\ &= XY + \bar{X}Z + XYZ + \bar{X}YZ \\ &= XY + XYZ + \bar{X}Z + \bar{X}YZ \\ &= XY(1 + Z) + \bar{X}Z(1 + Y) \\ &= XY + \bar{X}Z \end{aligned}$$

El dual del teorema de consenso es

$$(X + Y)(\bar{X} + Z)(Y + Z) = (X + Y)(\bar{X} + Z)$$

El ejemplo siguiente muestra cómo se puede aplicar el teorema de consenso durante la manipulación de una expresión booleana:

$$\begin{aligned} (A + B)(\bar{A} + C) &= A\bar{A} + AC + \bar{A}B + BC \\ &= AC + \bar{A}B + BC \\ &= AC + \bar{A}B \end{aligned}$$

Véase que $A\bar{A} = 0$ y $0 + AC = AC$. El término redundante eliminado por el teorema de consenso es BC.

El complemento de una función

La representación complementaria de una función F, $\bar{F}$, se obtiene de un intercambio de 1 por 0 y 0 por 1 en los valores de F en la tabla de verdad. El complemento de una función se puede derivar algebraicamente aplicando el Teorema de DeMorgan. La forma generalizada de este teorema expresa que se obtiene el complemento de una expresión mediante el intercambio de las operaciones AND y OR y complementando cada variable y cada constante, como se muestra en el Ejemplo 2-1.

EJEMPLO 2-1 Funciones de complemento

Encuentre el complemento de cada una de las funciones representadas por las ecuaciones $F_1 = \bar{X}YZ + \bar{X}\bar{Y}Z$ y $F_2 = X(\bar{Y}\bar{Z} + YZ)$. Aplicando el Teorema de DeMorgan tantas veces como sea necesario, obtenemos el complemento según lo siguiente:

$$\begin{aligned}\bar{F}_1 &= \overline{\bar{X}YZ + \bar{X}\bar{Y}Z} = \overline{(\bar{X}YZ)} \cdot \overline{(\bar{X}\bar{Y}Z)} \\ &= (X + \bar{Y} + Z)(X + Y + \bar{Z}) \\ \bar{F}_2 &= \overline{X(\bar{Y}\bar{Z} + YZ)} = \bar{X} + \overline{(\bar{Y}\bar{Z} + YZ)} \\ &= \bar{X} + (\overline{\bar{Y}\bar{Z}} \cdot \overline{YZ}) \\ &= \bar{X} + (Y + Z)(\bar{Y} + \bar{Z})\end{aligned}$$

■

Un método más simple para derivar el complemento de una función es calcular el dual de la ecuación de la función y complementar cada literal. Este método es el resultado de la generalización del Teorema de DeMorgan. Recuerde que se obtiene el dual de una expresión intercambiando las operaciones AND y OR y 1 y 0. Para evitar confusión en el manejo de funciones complejas, es útil añadir paréntesis alrededor de los términos antes de calcular el dual, según se ilustra en el siguiente ejemplo.

EJEMPLO 2-2 Complementando funciones usando dualidad

Encuentre los complementos de las funciones del Ejemplo 2-1 calculando los duales de sus ecuaciones y complementando cada literal.

Empezamos con

$$F_1 = \bar{X}YZ + \bar{X}\bar{Y}Z = (\bar{X}YZ) + (\bar{X}\bar{Y}Z)$$

El dual de F_1 es

$$(\bar{X} + Y + \bar{Z})(\bar{X} + \bar{Y} + Z)$$

Complementando cada literal, tenemos

$$(X + \bar{Y} + Z)(X + Y + \bar{Z}) = \bar{F}_1$$

Ahora,

$$F_2 = X(\bar{Y}\bar{Z} + YZ) = X((\bar{Y}\bar{Z}) + (YZ))$$

El dual de F_2 es

$$X + (\bar{Y} + \bar{Z})(Y + Z)$$

Complementando cada literal da lugar a

$$\bar{X} + (Y + Z)(\bar{Y} + \bar{Z}) = \bar{F}_2.$$

■

2-3 FORMAS CANÓNICAS

Se puede escribir una función booleana, expresada algebraicamente, de diferentes maneras. Hay, sin embargo, formas concretas de escribir las ecuaciones algebraicas que se consideran como formas canónicas. Las formas canónicas facilitan los procedimientos de simplificación para expresiones booleanas y frecuentemente da lugar a circuitos lógicos más deseables.

La forma canónica contiene *términos producto* y *términos suma*. Un ejemplo de un término producto es $X\bar{Y}Z$. Esto es un producto lógico formado por una operación AND de tres literales. Un ejemplo de un término suma es $X + Y + \bar{Z}$. Esto es una suma lógica formada por una operación OR entre los literales. Hay que darse cuenta de que las palabras «producto» y «suma» no implican operaciones aritméticas en el álgebra de Boole; en cambio, especifican las operaciones lógicas AND y OR respectivamente.

Minitérminos y maxitérminos

Se ha mostrado que la tabla de verdad define una función booleana. Una expresión algebraica que represente la función se puede derivar de la tabla buscando la suma lógica de todos los términos producto para los que la función asume el valor binario 1. A un término producto donde todas las variables aparecen exactamente una vez, sean complementadas o no complementadas, se le llama *minitérmino*. Su propiedad característica es que representa exactamente una combinación de las variables binarias en la tabla de verdad. Tiene el valor 1 para esta combinación y 0 para el resto. Hay 2^n diferentes minitérminos para n variables. Los cuatro minitérminos para las dos variables X e Y son y $\bar{X}\bar{Y}$, $\bar{X}Y$, $X\bar{Y}$ y XY. Los ocho minitérminos para las tres variables X, Y, y Z se muestran en la Tabla 2-6. Los números binarios de 000 a 111 se muestran debajo de las variables. Para cada combinación binaria hay un minitérmino asociado. Cada minitérmino es un término de producto de exactamente tres literales. Un literal es una variable complementada si el bit correspondiente de la combinación binaria asociada es 0 y es una variable no complementada si es 1. También se muestra un símbolo m_j para cada minitérmino en la tabla, donde el subíndice j denota el equivalente decimal de la combinación binaria para la que el minitérmino tiene el valor 1. Esta lista de minitérminos para cada n variables dadas se puede formar de manera similar a una lista de los números binarios de 0 a $2^n - 1$. Además, la tabla de verdad para cada minitérmino se muestra en la parte derecha de la tabla. Estas tablas de verdad muestran claramente que cada minitérmino es 1 para la combinación binaria correspondiente y 0 para todas las otras combinaciones. Más tarde, estas tablas de verdad serán útiles al usar minitérminos para formar expresiones booleanas.

A un término suma que contiene todas las variables de forma complementada o no complementada se le llama *maxitérmino*. Otra vez es posible formular 2^n maxitérminos con n variables. Los ocho maxitérminos para tres variables se muestran en la Tabla 2-7. Cada maxitérmino es una suma lógica de tres variables, donde cada variable se complementa si el bit correspondiente del número binario es 1 y no se complementa si es 0. El símbolo para un maxitérmino es M_j, donde j denota el equivalente decimal de una combinación binaria para la que el maxitérmino tiene el valor 0. En la parte derecha de la tabla, se muestra la tabla de verdad para cada maxitérmino. Véase que el valor del maxitérmino es 0 para la combinación correspondiente y 1 para el resto de combinaciones. Ahora está claro de donde salen los términos »minitérmino» y «maxitérmino»: un minitérmino es una función, no igual a 0, que tiene el menor número de 1s en su tabla de verdad; un maxitérmino es una función, no igual a 1, que tiene el mayor número de 1s en su tabla de verdad. Véase de la Tabla 2-6 y la Tabla 2-7 que los minitérminos y maxitérminos

□ **TABLA 2-6**
Minitérminos para tres variables

X	Y	Z	Término producto	Símbolo	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	$\bar{X}\bar{Y}\bar{Z}$	m_0	1	0	0	0	0	0	0	0
0	0	1	$\bar{X}\bar{Y}Z$	m_1	0	1	0	0	0	0	0	0
0	1	0	$\bar{X}Y\bar{Z}$	m_2	0	0	1	0	0	0	0	0
0	1	1	$\bar{X}YZ$	m_3	0	0	0	1	0	0	0	0
1	0	0	$X\bar{Y}\bar{Z}$	m_4	0	0	0	0	1	0	0	0
1	0	1	$X\bar{Y}Z$	m_5	0	0	0	0	0	1	0	0
1	1	0	$XY\bar{Z}$	m_6	0	0	0	0	0	0	1	0
1	1	1	XYZ	m_7	0	0	0	0	0	0	0	1

□ **TABLA 2-7**
Maxitérminos para tres variables

X	Y	Z	Término suma	Símbolo	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	0	0	$X + Y + Z$	M_0	0	1	1	1	1	1	1	1
0	0	1	$X + Y + \bar{Z}$	M_1	1	0	1	1	1	1	1	1
0	1	0	$X + \bar{Y} + Z$	M_2	1	1	0	1	1	1	1	1
0	1	1	$X + \bar{Y} + \bar{Z}$	M_3	1	1	1	0	1	1	1	1
1	0	0	$\bar{X} + Y + Z$	M_4	1	1	1	1	0	1	1	1
1	0	1	$\bar{X} + Y + \bar{Z}$	M_5	1	1	1	1	1	0	1	1
1	1	0	$\bar{X} + \bar{Y} + Z$	M_6	1	1	1	1	1	1	0	1
1	1	1	$\bar{X} + \bar{Y} + \bar{Z}$	M_7	1	1	1	1	1	1	1	0

con los mismos subíndices son los complementos entre sí; o sea, $M_j = \bar{m}_j$. Por ejemplo, para $j = 3$, tenemos

$$\bar{m}_3 = \overline{\bar{X}YZ} = X + \bar{Y} + \bar{Z} = M_3$$

Una función booleana puede ser representada algebraicamente por una tabla de verdad dada formando la suma lógica de todos los minitérminos que producen un 1 en la función. Esta expresión se llama una *suma de minitérminos*. Considere la función booleana F de la Tabla 2-8(a). La función es igual a 1 para cada una de las siguientes combinaciones binarias de las variables X, Y, y Z: 000, 010, 101 y 111. Esas combinaciones corresponden a los minitérminos 0, 2, 5 y 7. Examinando la Tabla 2-8 y las tablas de verdad para éstos minitérminos de la Tabla 2-6, es evidente que se puede expresar la función F algebraicamente como la suma lógica de los minitérminos formulados:

$$F = \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + X\bar{Y}Z + XYZ = m_0 + m_2 + m_5 + m_7$$

□ **TABLA 2-8**
Funciones booleanas de tres variables

(a) X	Y	Z	F	$\bar{F}$	(b) X	Y	Z	E
0	0	0	1	0	0	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	1	0	0	1	0	1
0	1	1	0	1	0	1	1	0
1	0	0	0	1	1	0	0	1
1	0	1	1	0	1	0	1	1
1	1	0	0	1	1	1	0	0
1	1	1	1	0	1	1	1	0

Esto se puede abreviar más enumerando solamente los subíndices decimales de los minitérminos:

$$F(X, Y, Z) = \Sigma m(0, 2, 5, 7)$$

El símbolo Σ significa la suma lógica (OR booleana) de los minitérminos. Los números en paréntesis representan los minitérminos de la función. Las letras entre paréntesis que van a continuación de F forman una lista de variables en el orden de conversión de los minitérminos a términos producto.

Ahora considere el complemento de una función booleana. Los valores binarios de $\bar{F}$ de la Tabla 2-8(a) se obtienen cambiando 1 a 0 y 0 a 1 en los valores de F. Partiendo de la suma lógica de los minitérminos de $\bar{F}$, obtenemos

$$\bar{F} = \bar{X}\bar{Y}Z + \bar{X}YZ + X\bar{Y}\bar{Z} + XY\bar{Z} = m_1 + m_3 + m_4 + m_6$$

o, de forma abreviada,

$$\bar{F}(X, Y, Z) = \Sigma m(1, 3, 4, 6)$$

Véase que los números de los minitérminos de $\bar{F}$ son los que faltan en la lista de números de los minitérminos de F. Ahora tomamos el complemento de para obtener F:

$$\begin{aligned} F &= \overline{m_1 + m_3 + m_4 + m_6} = \overline{m_1} \cdot \overline{m_3} \cdot \overline{m_4} \cdot \overline{m_6} \\ &= M_1 \cdot M_3 \cdot M_4 \cdot M_6 \text{ (ya que } \overline{m_j} = M_j) \\ &= (X + Y + \bar{Z})(X + \bar{Y} + \bar{Z})(\bar{X} + Y + Z)(\bar{X} + \bar{Y} + Z) \end{aligned}$$

Esto muestra el procedimiento para expresar una función booleana como *producto de maxitérminos*. La forma abreviada para ese producto es

$$F(X, Y, Z) = \Pi M(1, 3, 4, 6)$$

donde el símbolo Π denota el producto lógico (AND booleana) de los maxitérminos cuyos números se enumeran entre paréntesis. Véase que los números decimales incluidos en el producto de maxitérminos siempre serán los mismos que los de la lista de minitérminos de la función complementada, como (1, 3, 4, 6) del ejemplo anterior. Los maxitérminos se usan rara vez cuando se trata con funciones booleanas, porque es siempre posible reemplazarlos con la lista de minitérminos de $\bar{F}$.

A continuación se resumen las propiedades más importantes de los minitérminos:

1. Hay 2^n minitérminos para n variables booleanas. Estos minitérminos se pueden evaluar a partir de los números binarios de 0 a $2^n - 1$.
2. Cada función booleana se puede expresar como suma lógica de minitérminos.
3. El complemento de una función contiene los minitérminos que no están incluidos en la función original.
4. Una función que incluye todos los 2^n minitérminos es igual a un 1 lógico.

Una función que no tiene la forma de suma de minitérminos puede convertirse a esta forma mediante la tabla de verdad, mientras que la tabla de verdad especifica los minitérminos de la función. Considere, por ejemplo, la función booleana

$$E = \bar{Y} + \bar{X}\bar{Y}$$

La expresión no tiene forma de suma de minitérminos, porque cada término no contiene todas los tres variables *X*, *Y*, y *Z*. En la Tabla 2-8(b) se enumera la tabla de verdad de esta función. De la tabla, obtenemos los minitérminos de la función:

$$E(X, Y, Z) = \Sigma m(0, 1, 2, 4, 5)$$

Los minitérminos para el complemento de *E* resultan de

$$\bar{E}(X, Y, Z) = \Sigma m(3, 6, 7)$$

Véase que el número total de minitérminos en *E* y $\bar{E}$ es igual a ocho, ya que la función tiene tres variables, y tres variables producen un total de ocho minitérminos. Con cuatro variables habrá un total de 16 minitérminos, y para dos variables, habrá 4 minitérminos. Un ejemplo de una función que incluye todos los minitérminos es

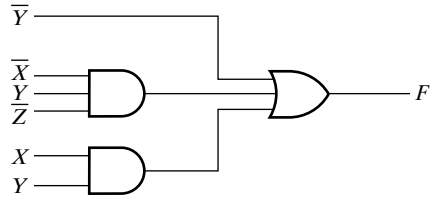
$$G(X, Y) = \Sigma m(0, 1, 2, 3) = 1$$

Como *G* es una función de dos variables y contiene todos los cuatro minitérminos, siempre es igual a un 1 lógico.

Suma de productos

La forma de suma de minitérminos es una expresión algebraica canónica que se obtiene directamente de una tabla de verdad. La expresión obtenida de esta manera contiene el número máximo de literales en cada término y tiene normalmente más productos de los que son necesarios. La razón para esto es que, por definición, cada minitérmino tiene que incluir todas las variables de la función, complementada o no complementada. Si se ha obtenido una vez la suma de minitérminos de la tabla de verdad, el siguiente paso es intentar simplificar la expresión para ver si es posible reducir el número de productos y el número de literales en los términos. El resultado es una expresión simplificada en la forma de *suma de productos*. Esto es una forma canónica alternativa de expresión que contiene productos con uno, dos, o cualquier número de literales. Un ejemplo de una función booleana expresada como suma de productos es

$$F = \bar{Y} + \bar{X}Y\bar{Z} + XY$$



□ **FIGURA 2-5**
Implementación con suma de productos

La expresión tiene tres productos, el primero con un literal, el segundo con tres literales, y el tercero con dos literales.

El diagrama lógico para una forma de suma de productos está formado por un grupo de puertas AND seguido de una única puerta OR, como se muestra en la Figura 2-5. Cada producto requiere una puerta AND, excepto para el término con un único literal. La suma lógica se forma con una puerta OR que tiene como entradas literales únicos y la salida de puertas AND. Se supone que las variables de entrada están directamente disponibles en sus formas complementadas y no complementadas, así que no se incluyen inversores en el diagrama. Las puertas AND seguidas por la puerta OR forman una configuración de circuito al que se le denomina como una *implementación de dos niveles* o como *circuito de dos niveles*.

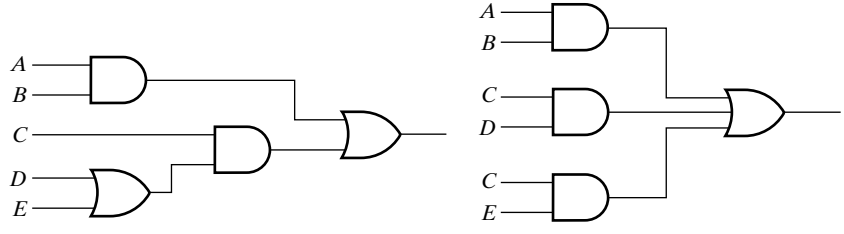
Si una expresión no está en la forma de suma de productos, se puede convertir a una forma canónica mediante las leyes distributivas. Considere la expresión

$$F = AB + C(D + E)$$

Esta no está en forma de suma de productos, porque el término $D + E$ es parte de un producto, pero no es un literal único. La expresión puede convertirse en una suma de productos aplicando la ley distributiva apropiada, como se muestra a continuación:

$$F = AB + C(D + E) = AB + CD + CE$$

En la Figura 2-6(a) se ve la función F implementada en forma no canónica. Esto requiere dos puertas AND y dos puertas OR. Hay tres niveles de puertas en el circuito. En la Figura 2-6(b), se ha implementado F en forma de suma de productos. Este circuito requiere tres puertas AND y una puerta OR y usa dos niveles de puertas. La decisión de usar una implementación de dos o de múltiples niveles (tres o más) es compleja. Los problemas aquí involucrados son el número de puertas, el número de entradas a las puertas y el retardo entre el momento en que las variables de entrada están puestas y el momento en que aparecen los valores resultantes en la



(a) $AB + C(D + E)$ (b) $AB + CD + CE$

□ **FIGURA 2-6**
Implementación de tres y dos niveles

salida. Las implementaciones de dos niveles son la forma natural para ciertas tecnologías, como veremos en el Capítulo 4.

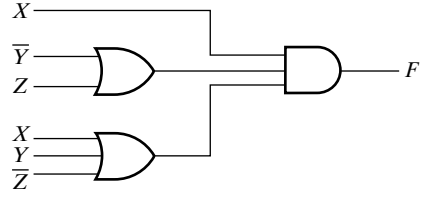
Producto de sumas

Otra forma canónica para expresar funciones booleanas algebraicamente es el *producto de sumas*. Esta forma se obtiene formando un producto lógico de sumas. Cada término de la suma lógica puede tener cualquier número de literales diferentes. Un ejemplo de una función expresada de forma de suma de productos es

$$F = X(\overline{Y} + Z)(X + Y + \overline{Z})$$

Esta expresión tiene sumas de uno, dos y tres literales. Los términos de suma realizan una operación OR, y el producto es una operación AND.

La estructura de las puertas de la expresión de productos de suma esta formada por un grupo de puertas OR para las sumas (excepto para el término con un único literal), seguido de una puerta AND. Esto se muestra en la Figura 2-7 para la anterior función *F*. Como en el caso de suma de productos, este tipo de expresión canónica está formada por una estructura de dos niveles de puertas.



□ **FIGURA 2-7**
Implementación de producto de sumas

2-4 OPTIMIZACIÓN DE CIRCUITOS DE DOS NIVELES

La complejidad de las puertas lógicas digitales que realizan una función booleana está relacionada directamente con la expresión algebraica a partir de la cual se implementa la función. Aunque la representación de una función en la tabla de verdad es única, cuando se expresa algebraicamente, la función puede aparecer en muchas formas diferentes. Las expresiones booleanas podrían simplificarse mediante manipulación algebraica como se ha discutido en la Sección 2-2. Sin embargo, este procedimiento de simplificación es malo porque carece de reglas especiales para predecir cada paso que ocurre en el proceso manipulativo y es difícil de determinar si se ha conseguido la expresión más sencilla. Por otro lado, el método del mapa provee un procedimiento directo para optimizar funciones booleanas con un máximo de cuatro variables. Se puede pintar también mapas para cinco y seis variables, pero estas son más incómodas de usar. El mapa se conoce también como *Mapa de Karnaugh*, o *mapa-K*. El mapa es un diagrama hecho con cuadrados, donde cada cuadrado representa un minitérmino de la función. Puesto que cualquier función booleana se puede expresar como suma de minitérminos, una función booleana puede ser reconocida gráficamente en el mapa por aquellos cuadrados cuyos minitérminos se incluyen en la función. De hecho, el mapa presenta un diagrama visual de todos los caminos posibles para expresar una función en forma canónica. Reconociendo diferentes patrones, el usuario puede derivar expresiones algebraicas alternativas para la misma función, de las cuales

se selecciona la más sencilla. Las expresiones optimizadas producidas por el mapa están siempre en forma de suma de productos o producto de sumas. Así, los mapas manejan la optimización para implementaciones de dos niveles, pero no se puede aplicar directamente a posibles implementaciones más sencillas para el caso general con tres o más niveles. Inicialmente, esta sección cubre la optimización de suma de productos y, más tarde, aplica la optimización de producto de sumas.

Criterios de coste

En la sección anterior, el número de literales y términos se vio como una manera de medir la simplicidad de un circuito lógico. Ahora introducimos dos criterios de coste para formalizar este concepto.

El primer criterio es el *coste por literal*, el número de veces que aparecen los literales en una expresión booleana que corresponde exactamente al diagrama lógico. Por ejemplo, para los circuitos de la Figura 2-6, Las expresiones booleanas correspondientes son

$$F = AB + C(D + E) \quad \text{y} \quad F = AB + CD + CE$$

En la primera ecuación aparecen cinco literales y seis en la segunda, de esta forma, la primera ecuación es la más simple en términos de coste por literal. El coste por literal tiene la ventaja que se puede evaluar muy sencillamente contando la aparición de los literales. Sin embargo, no representa correctamente la complejidad del circuito en todos los casos, ni siquiera para la comparación de diferentes implementaciones de la misma función lógica. Las siguientes ecuaciones booleanas, ambas de la función G, muestran esta situación:

$$G = ABCD + \bar{A}\bar{B}\bar{C}D \quad \text{y} \quad G = (\bar{A} + B)(\bar{B} + C)(\bar{C} + D)(\bar{D} + A)$$

Las implementaciones representadas por esas ecuaciones tienen ambas un coste literal de ocho. Pero, la primera ecuación tiene dos términos y la segunda tiene cuatro. Esto sugiere que la primera ecuación tiene un coste más bajo que la segunda.

Para entender la diferencia ilustrada, definimos el *coste de entradas de puerta* como el número de entradas a las puertas en la implementación que corresponde exactamente a la ecuación o las ecuaciones dadas. Este coste se puede determinar fácilmente a partir del diagrama lógico contando simplemente el número total de entradas a las puertas. Para las ecuaciones de suma de productos o producto de sumas, se puede averiguar encontrando la suma de

(1) todas las apariciones de los literales,
(2) el número de términos excluyendo términos que solamente consisten en un único literal, y, opcionalmente,
(3) el número de diferentes literales complementados.

En (1), se representan todas las entradas de las puertas desde fuera del circuito. En (2), se representa todas las entradas de las puertas dentro del circuito, excepto las que van a los inversores y en (3), los inversores necesarios para complementar las variables de entrada, que se cuentan en el momento que no se proporcionan las variables de entrada complementadas. Para las dos ecuaciones precedentes, excluyendo la suma de (3), las respectivas sumas de las entradas de las puertas son $8 + 2 = 10$ y $8 + 4 = 12$. Incluyendo la suma de (3), la de inversores de entrada, las sumas respectivas son 14 y 16. Así, la primera ecuación para G tiene un coste por entradas más bajo, aunque los costes por literales sean iguales.

El coste por entradas de puerta es ahora una buena medida para implementaciones lógicas actuales ya que es proporcional el número de transistores y conexiones usadas para implementar un circuito lógico. La representación de las entradas de las puertas va ser particularmente más importante para medir los costes de los circuitos con más que dos niveles. Típicamente, como el número de niveles incrementa, el coste literal representa una proporción más pequeña en el coste de los circuitos actuales, ya que más y más puertas no tienen entradas desde fuera del mismo circuito. Más adelante, en la Figura 2-29, introducimos otros tipos de puertas complejas para las que la evaluación del coste de las entradas de las puertas a partir de una ecuación no es válida, ya que la correspondencia entre las operaciones AND, OR y NOT de la ecuación y las puertas del circuito no se puede establecer. En estos casos, tanto como para ecuaciones más complejas que las sumas de productos y los productos de sumas, hay que determinar la suma de las entradas de las puertas directamente desde la implementación.

Sin tener en cuenta los criterios de coste usados, más adelante veremos que la expresión más sencilla no es necesariamente la única. A veces es posible encontrar dos o más expresiones que cumplen el criterio de coste aplicado. En este caso, cada solución es satisfactoria desde el punto de vista de coste.

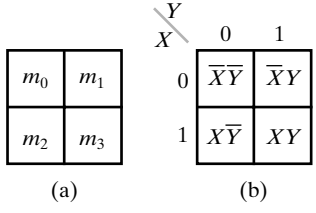
Mapa de dos variables

Hay cuatro minitérminos en una función booleana con dos variables. Así, el mapa de dos variables consiste en cuatro cuadrados, uno por cada minitérmino, según se muestra en la Figura 2-8(a). El mapa se muestra otra vez en la Figura 2-8(b) para señalar la relación entre los cuadrados y las dos variables X e Y. El 0 y 1 marcado, en el lado izquierdo y en la parte superior del mapa indican los valores de las variables. La variable X aparece complementada en la fila 0 y no complementada en la fila 1. De igual manera, Y aparece complementada en la columna 0 y no complementada en la columna 1. Véase que las cuatro combinaciones de estos valores binarios corresponden a las filas de la tabla de verdad asociadas a los cuatro minitérminos.

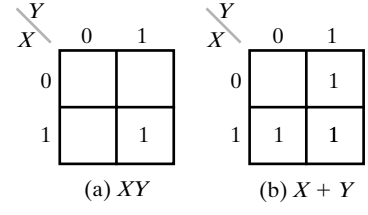
Una función de dos variables puede ser representada en un mapa marcando los cuadrados que corresponden a los minitérminos de la función. Como ejemplo, se muestra la función XY de la Figura 2-9(a). Ya que XY es igual al minitérmino m_3, se pone un 1 dentro del cuadrado que pertenece a m_3. En la Figura 2-9(b) se muestra el mapa para la suma lógica de tres minitérminos:

$$m_1 + m_2 + m_3 = \bar{X}Y + X\bar{Y} + XY = X + Y$$

La expresión optimizada $X + Y$ se determina del área de dos cuadrados para la variable X en la segunda fila y del área de dos cuadrados para Y en la segunda columna. Juntas, estas dos áreas



□ **FIGURA 2-8**
Mapa de dos variables



□ **FIGURA 2-9**
Representación de funciones en el mapa

incluyen los tres cuadrados pertenecientes a X o Y. Esta simplificación puede justificase mediante manipulación algebraica:

$$\bar{X}Y + X\bar{Y} + XY = \bar{X}Y + X(\bar{Y} + Y) = (\bar{X} + X)(Y + X) = X + Y$$

El procedimiento exacto para combinar cuadrados en el mapa se va a aclarar en los siguientes ejemplos.

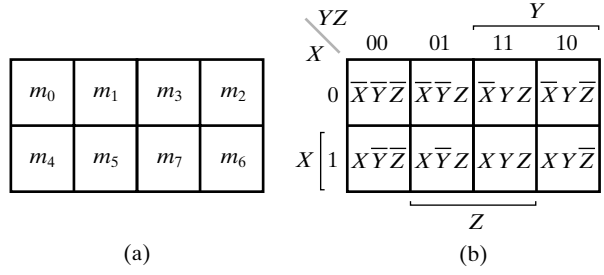
Mapa de tres variables

Hay ocho minitérminos para tres variables binarias. Por eso, un mapa de tres variables tiene ocho cuadrados, como se indica en la Figura 2-10. El mapa dibujado en la parte (b) está marcado con números binarios para cada fila y cada columna para mostrar los valores binarios de los minitérminos. Note que los números en las columnas no siguen la secuencia de la cuenta binaria. La característica de la secuencia enumerada es que sólo un bit cambia su valor de una columna hacia la adyacente, que corresponde al Código Gray introducido en el Capítulo 1.

Un cuadrado perteneciente a un minitérmino se puede localizar en el mapa de dos maneras. Primero, podemos memorizar los números enumerados en la Figura 2-10(a) para cada lugar del minitérmino, o podemos referirnos a los números binarios dentro de las filas y columnas de la Figura 2-10(b). Por ejemplo, el cuadrado asignado a m_5 corresponde a la fila 1 y a la columna 01. Cuando se combinan estos dos números, dan lugar al número binario 101, cuyo equivalente decimal es 5.

Otra posibilidad de localizar el cuadrado $m_5 = X\bar{Y}Z$ es verlo situado en la fila marcada con X y la columna que pertenece a $\bar{Y}Z$ (columna 01). Véase que hay cuatro cuadrados donde cada variable es igual a 1 y cuatro donde cada una es igual a 0. La variable aparece sin complementar en los cuatro cuadrados donde es igual a 1 y de forma complementada en los cuatro cuadrados donde es igual a 0. Es más conveniente escribir el nombre de la variable al lado de los cuatro cuadrados donde no aparece complementada. Después de familiarizarse con los mapas, el uso de los nombres de los variables es suficiente para identificar las regiones del mapa. Para esto, es importante localizar estas etiquetas para obtener todos los minitérminos del mapa.

En el mapa de dos variables, la función XY mostraba que una función o un término de una función pueden estar formados por un único cuadrado del mapa. Pero para conseguir una simplificación, hay que considerar varios cuadrados que corresponden a términos productos. Para entender cómo la combinación de cuadrados simplifica las funciones booleanas, tenemos que conocer la propiedad básica de cuadrados adyacentes: cada pareja de cuadrados adyacentes, horizontales o verticales (pero no diagonales), que forman un rectángulo, corresponden a minitér-



(a) (b)

□ **FIGURA 2-10**
Mapa de tres variables

minos que varían en una sola variable. Esta variable aparece sin complementar en un cuadrado y complementada en el otro. Por ejemplo, m_5 y m_7 están situadas en dos cuadrados adyacentes. Se encuentra el complemento de Y en m_5 y la misma variable, sin complementar, en m_7, mientras las otras dos variables son iguales en los dos cuadrados. La suma lógica de dos de estos minitérminos adyacentes se puede simplificar en un único término producto de dos variables:

$$m_5 + m_7 = X\bar{Y}Z + XYZ = XZ(\bar{Y} + Y) = XZ$$

Aquí, los dos cuadrados son diferentes en cuanto a la variable Y, que se puede quitar cuando se calcula la suma lógica (OR) de los minitérminos. Así, en un mapa de 3 variables, cada dos minitérminos en cuadrados adyacentes que se combinan con una OR producen un término producto de dos variables. Esto se muestra en el Ejemplo 2-3.

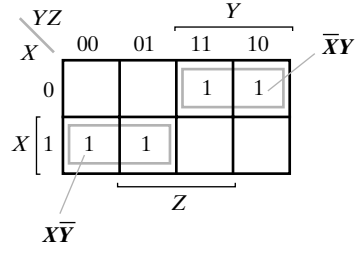
EJEMPLO 2-3 Simplificación de una función booleana usando un mapa

Simplifique la función booleana

$$F(X, Y, Z) = \Sigma m(2, 3, 4, 5)$$

Primero, se pone un 1 en cada minitérmino que representa la función. Esto se muestra en la Figura 2-11, donde los cuadrados para los minitérminos 010, 011, 100, y 101 se han rellenado con 1s. Es mejor dejar todos los cuadrados en los que la función tiene el valor 0 en blanco en vez de poner los 0s. El paso siguiente es encontrar grupos de cuadrados en el mapa que representan términos producto considerados para la expresión simplificada. Llamamos a estos objetos *rectángulos*, ya que su forma es un rectángulo (incluyendo, por supuesto, un cuadrado). Sin embargo, los rectángulos que corresponden a términos producto están restringidos al contener un número de cuadrados que son potencias de 2, como 1, 2, 4, 8, ... Así, nuestro objetivo es encontrar el menor número de rectángulos que incluyan a todos los minitérminos marcados con 1s. Esto va a dar lugar a un mínimo de términos producto. En el mapa de la figura, los dos rectángulos acogen a los cuatro cuadrados que contienen 1. El rectángulo superior derecho representa al término producto $\bar{X}Y$. Esto se determina observando que el rectángulo está en la fila 0, que corresponde a $\bar{X}$, y las últimas dos columnas, correspondiendo a Y. De igual manera, el rectángulo inferior izquierdo representa el término de producto $X\bar{Y}$. (La segunda fila representa X y las dos columnas a la izquierda representan $\bar{Y}$.) Ya que estos dos rectángulos incluyen a todos de los 1s del mapa, la suma lógica de los dos términos producto correspondientes dan como resultado una expresión optimizada de F:

$$F = \bar{X}Y + X\bar{Y}$$



□ **FIGURA 2-11**
Mapa para Ejemplo 2-3: $F(X, Y, Z) = \Sigma m(2, 3, 4, 5) = \bar{X}Y + X\bar{Y}$ ■

En algunos casos, dos cuadrados del mapa son adyacentes y forman un rectángulo de tamaño dos, aunque no se tocan. Por ejemplo, en la Figura 2-10, m_0 es adyacente a m_2 y m_4 es adyacente a m_6 porque los minitérminos se distinguen en una variable. Esto se puede verificar algebraicamente:

$$m_0 + m_2 = \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} = \bar{X}\bar{Z}(\bar{Y} + Y) = \bar{X}\bar{Z}$$
$$m_4 + m_6 = X\bar{Y}\bar{Z} + XY\bar{Z} = X\bar{Z}(\bar{Y} + Y) = X\bar{Z}$$

Los rectángulos que corresponden a estos dos términos producto, $\bar{X}\bar{Z}$ y $X\bar{Z}$, se muestran en el mapa de la Figura 2-12(a). Basándose en la ubicación de estos rectángulos, tenemos que modificar la definición de los cuadrados adyacentes para incluir este y otros casos similares. Lo hacemos considerando el mapa como si estuviese dibujado en forma de cilindro, como se muestra en la Figura 2-12(b), donde los bordes derechos e izquierdos se tocan para establecer correctamente las vecindades de los minitérminos y formar los rectángulos. En los mapas de la Figura 2-12, hemos usado simplemente números en vez de *m* para representar los minitérminos. Cualquiera de estas notaciones se usará libremente.

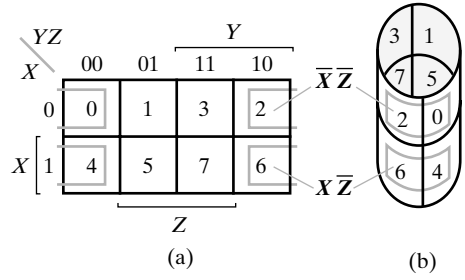
Un rectángulo de cuatro cuadrados representa un término producto que es la suma lógica de cuatro minitérminos. Para el caso de tres variables, un término producto de estos está formado por un solo literal. Como ejemplo, la suma lógica de cuatro minitérminos adyacentes 0, 2, 4, y 6 se reducen a un único término literal $\bar{Z}$:

$$\begin{aligned} m_0 + m_2 + m_4 + m_6 &= \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XY\bar{Z} \\ &= \bar{X}\bar{Z}(\bar{Y} + Y) + X\bar{Z}(\bar{Y} + Y) \\ &= \bar{X}\bar{Z} + X\bar{Z} = \bar{Z}(\bar{X} + X) = \bar{Z} \end{aligned}$$

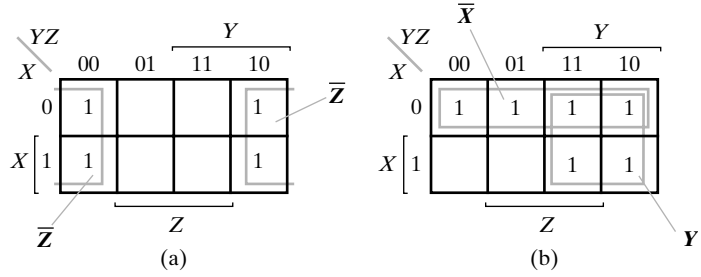
El rectángulo para este término producto se muestra en la Figura 2-13(a). Véase que el término producto se basa en que los bordes izquierdos y derechos del mapa son adyacentes de manera que forman un rectángulo. Los otros dos ejemplos de rectángulos que se corresponden con términos producto derivados de cuatro minitérminos se muestran en la Figura 2-13(b).

En general, ya que se combinan más cuadrados, obtenemos un término producto con menos literales. Los mapas de tres variables requieren las siguientes características:

Un cuadrado representa un minitérmino de tres literales.
Un rectángulo de dos cuadrados representa un término de producto de dos literales.
Un rectángulo de cuatro cuadrados representa un término de producto de un literal.
Un rectángulo de ocho cuadrados abarca el mapa entero y produce una función que es siempre igual a 1 lógico.
Éstas características se muestran en el Ejemplo 2-4.



□ **FIGURA 2-12**
Mapa de tres variables: Plano y en cilindro para mostrar los cuadrados adyacentes



□ **FIGURA 2-13**
Términos producto usando cuatro minitérminos

EJEMPLO 2-4 Simplificación de funciones de tres variables con mapas

Simplifique las dos funciones booleanas siguientes:

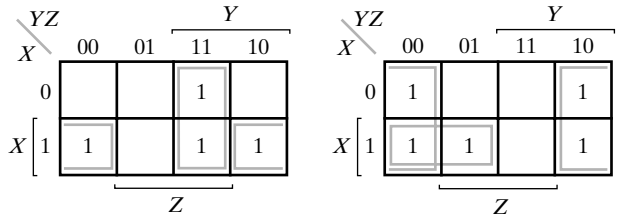
$$F_1(X, Y, Z) = \Sigma m(3, 4, 6, 7)$$
$$F_2(X, Y, Z) = \Sigma m(0, 2, 4, 5, 6)$$

El mapa de F_1 se muestra en la Figura 2-14(a). Hay cuatro cuadrados marcados con 1s, uno para cada minitérmino de la función. Se combinan dos cuadrados adyacentes en la tercera columna para llegar al término de dos literales YZ. Los dos cuadrados sobrantes, también con 1, son adyacentes por la definición basada en el cilindro y se muestran en el diagrama con sus valores incluidos en medios rectángulos. Cuando se combinan, estos dos cuadrados construyen el término de dos literales $X\bar{Z}$. Así la función optimizada pasa a ser

$$F_1 = YZ + X\bar{Z}$$

El mapa para F_2 se muestra en la Figura 2-14(b). Primero, combinamos los cuatro cuadrados adyacentes en las primeras y últimas columnas, basándonos en lo que hemos aprendido de la Figura 2-13, para llegar al término de un literal $\bar{Z}$. El último cuadrado sobrante que representa el minitérmino 5 se combina con un cuadrado adyacente que ya ha sido usado una vez. Esto no sólo se permite, sino que es deseable, ya que los dos cuadrados adyacentes llegan al término de dos literales $X\bar{Y}$, con el cuadrado simple representando el minitérmino de tres literales $XY\bar{Z}$. La función optimizada es

$$F_2 = \bar{Z} + X\bar{Y}$$



(a) $F_1(X, Y, Z) = \Sigma m(3, 4, 6, 7) = YZ + X\bar{Z}$ (b) $F_2(X, Y, Z) = \Sigma m(0, 2, 4, 5, 6) = \bar{Z} + X\bar{Y}$

□ **FIGURA 2-14**
Mapas para el Ejemplo 2-4

■

Existen en ocasiones formas alternativas de combinar cuadrados para producir expresiones igualmente optimizadas. Un ejemplo de esto se muestra en el mapa de la Figura 2-15. Los minitérminos 1 y 3 se combinan para llegar al término $\bar{X}Z$, y los minitérminos 4 y 6 producen el término $X\bar{Z}$. Sin embargo, hay dos formas de combinar el cuadrado del minitérmino 5 con otro cuadrado adyacente para producir un tercer término de dos literales. Combinándolo con el minitérmino 4 se llega al término $X\bar{Y}$; pero combinándolo con el minitérmino 1 se llega al término $\bar{Y}Z$. Cada una de las dos expresiones, enumeradas en la Figura 2-15, tienen tres términos de dos literales cada uno, de esta forma, hay dos posibles soluciones optimizadas para esta función.

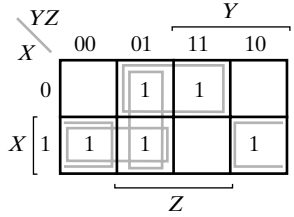
Si una función no se expresa como suma de minitérminos, podemos usar el mapa para obtener los minitérminos de la función y después simplificarla. Sin embargo, es necesario tener la expresión algebraica en forma de suma de productos, a partir de la cual cada término producto se dibuja en el mapa. Como ejemplo considere la función booleana

$$F = \bar{X}Z + \bar{X}Y + X\bar{Y}Z + YZ$$

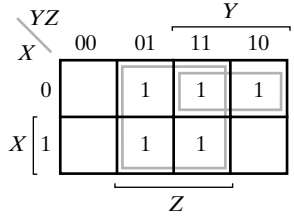
Los tres términos producto de la expresión tienen dos literales y están representados en un mapa de tres variables por dos cuadrados cada uno. Los dos cuadrados correspondientes al primer término, $\bar{X}Z$, se encuentran en la Figura 2-16 donde coinciden $\bar{X}$ (primera columna) y Z (dos columnas medianas), dando lugar a 1 en los cuadrados 001 y 011. Véase que cuando se marcan los 1 en los cuadrados, es posible encontrar un 1 ya puesto por el término precedente. Esto pasa con el segundo término, $\bar{X}Y$, que tiene 1 en los cuadrados 011 y 010; pero el cuadrado 011 lo tiene en común con el primer término, $\bar{X}Z$, así se marca solamente un 1 en él. Continuando de esta manera, encontramos que la función tiene cinco minitérminos, como se indica por los cinco 1 de la figura. Los minitérminos se leen directamente a partir del mapa para ser 1, 2, 3, 5, y 7. La función originalmente dada tiene cuatro términos producto. Puede ser optimizada en el mapa con solamente dos únicos términos

$$F = Z + \bar{X}Y$$

con lo que resulta una reducción significativa del coste de la implementación.



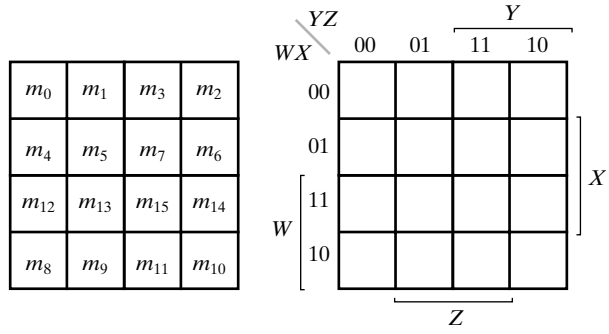
□ **FIGURA 2-15**
$F(X, Y, Z) = \Sigma m(1, 3, 4, 5, 6)$
$= \bar{X}Z + X\bar{Z} + X\bar{Y}$
$= \bar{X}Z + X\bar{Z} + \bar{Y}Z$



□ **FIGURA 2-16**
$F(X, Y, Z) = \Sigma m(1, 2, 3, 5, 7) = Z + \bar{X}Y$

Mapa de cuatro variables

Hay 16 minitérminos para cuatro variables binarias, y por esto, un mapa de cuatro variables está formado por 16 cuadrados, como se indica en la Figura 2-17. La asignación de minitérminos en cada cuadrado se indica en la parte (a) del diagrama. Se dibuja el mapa en (b) otra vez para mostrar la relación de las cuatro variables. Las filas y columnas se enumeran de manera que



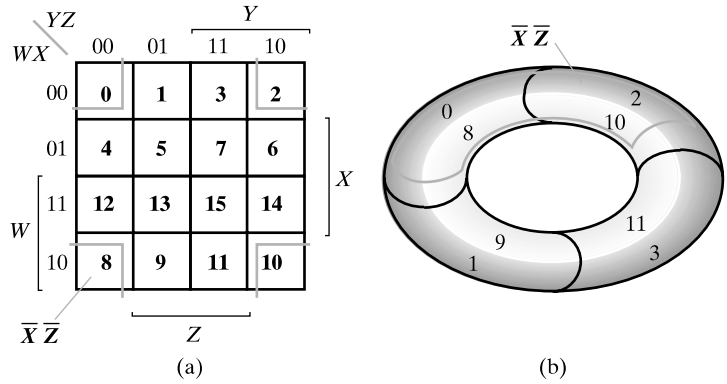
□ **FIGURA 2-17**
Mapa de cuatro variables

sólo un bit del número binario cambia su valor entre cada dos columnas o filas adyacentes, garantizando la misma propiedad para los cuadrados adyacentes. Los números de las filas y las columnas corresponden a un Código Gray de dos bits, como se introdujo en Capítulo 1. Los minitérminos correspondientes a cada cuadrado se pueden obtener combinando los números de la fila y la columna. Por ejemplo, combinando los números de la tercera fila (11) y la segunda columna (01) se obtiene el número binario 1101, el equivalente binario de 13. Así, el cuadrado en la tercera fila, segunda columna, representa el minitérmino m_{13}. Además, se marca cada variable en el mapa para mostrar los ocho cuadrados donde aparecen sin complementar. Los otros ocho cuadrados, donde no se indica ningúna etiqueta, corresponden a la variable que se complementa. Así, *W* aparece complementado en las dos primeras líneas y sin complementar en las dos segundas filas.

El método usado para simplificar funciones de cuatro variables es similar al que se usa para simplificar funciones de tres variables. Como cuadrados adyacentes se definen los que se encuentran uno al lado de otro, como para los mapas de dos y tres variables. Para mostrar las vecindades entre cuadrados, se ha dibujado el mapa de la Figura 2-18(a) como un toro en la Figura 2-18(b), con los bordes superior e inferior, como también los bordes derecho e izquierdo, tocándose mutuamente para mostrar cuadrados adyacentes. Por ejemplo, m_0 y m_2 son dos cuadrados adyacentes, como lo son también m_0 y m_8. Las combinaciones elegibles posibles durante el proceso de optimización en el mapa de cuatro variables son las siguientes:

Un cuadrado representa un minitérmino de cuatro literales.
Un rectángulo de 2 cuadrados representa un término producto de tres literales.
Un rectángulo de 4 cuadrados representa un término producto de dos literales.
Un rectángulo de 8 cuadrados representa un término producto de un literal.
Un rectángulo de 16 cuadrados produce una función que es siempre igual a 1 lógico.

No se puede usar ninguna otra combinación de cuadrados. Un término producto interesante de dos literales, $\overline{X} \cdot \overline{Z}$, se muestra en la Figura 2-18. En (b), donde se ve el mapa como un toro, las vecindades de los cuadrados que representan este término producto quedan claras, pero en (a) estos cuadrados están en las cuatro esquinas del mapa y así aparecen muy lejanos uno de otro. Es importante recordar este término producto, ya que se olvida muchas veces. También sirve como recordatorio de que los bordes izquierdo y derecho del mapa son adyacentes, tanto como los bordes superior e inferior. Así, los rectángulos del mapa cruzan los bordes derecho e izquierdo, superior e inferior, o ambos.



□ **FIGURA 2-18**
Mapa de cuatro variables: Plano y toro para mostrar las vecindades

Los siguientes ejemplos muestran el procedimiento de simplificar funciones booleanas de cuatro variables.

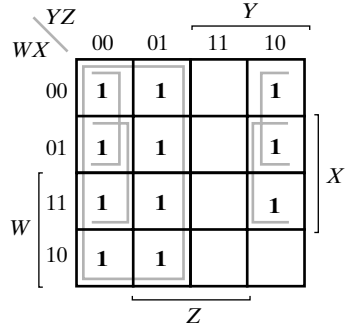
EJEMPLO 2-5 Simplificación de una función de 4 variables mediante un mapa

Simplifique la función booleana

$$F(W, X, Y, Z) = \Sigma m(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$

Los minitérminos de la función se han marcado con 1 en el mapa de la Figura 2-19. Los ocho cuadrados de las dos columnas a la izquierda se combinan para formar un rectángulo para el término literal único, $\bar{Y}$. Los tres 1s sobrantes no se pueden combinar para llegar a un término simplificado; más bien, se tienen que combinar como rectángulos de dos o cuatro cuadrados. Los dos 1s superiores de la derecha se combinan con los dos 1s superiores de la izquierda para dar lugar al término $\bar{W}\bar{Z}$. Véase otra vez que está permitido usar el mismo cuadrado más de una vez. Ahora nos queda un cuadrado marcado con 1 en la tercera fila y cuarta columna (minitérmino 1110). En vez de tomar este cuadrado sólo, lo que da lugar a un término de cuatro literales, lo combinamos con cuadrados ya usados para formar un rectángulo de cuatro cuadrados en las dos filas intermedias y las dos columnas finales, resultando el término $X\bar{Z}$. La expresión optimizada es la suma lógica de los tres términos:

$$F = \bar{Y} + \bar{W}\bar{Z} + X\bar{Z}$$



□ **FIGURA 2-19**
Mapa para el Ejemplo 2-5: $F = \bar{Y} + \bar{W}\bar{Z} + X\bar{Z}$

■

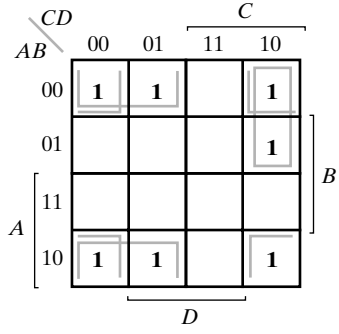
EJEMPLO 2-6 Simplificación de una función de cuatro variables mediante un mapa

Simplifique la función booleana

$$F = \bar{A}\bar{B}\bar{C} + \bar{B}C\bar{D} + A\bar{B}\bar{C} + \bar{A}BC\bar{D}$$

Esta función tiene cuatro variables: *A*, *B*, *C*, y *D*. Se expresa en forma de suma de productos con tres términos de tres literales cada y un término de cuatro literales. El área del mapa cubierto por la función se muestra en la Figura 2-20. Cada término de tres literales se representa en el mapa por dos cuadrados. $\bar{A}\bar{B}\bar{C}$ se representa por los cuadrados 0000 y 0001, $\bar{B}C\bar{D}$ por los cuadrados 0010 y 1010, y $A\bar{B}\bar{C}$ por los cuadrados 1000 y 1001. El término con cuatro literales es el minitérmino 0110. La función está simplificada en el mapa tomando los 1s de las cuatro esquinas, para llegar al término $\bar{B}\bar{D}$. Este término producto está en los mismos sitios del mapa que $\bar{X}\bar{Z}$ de la Figura 2-18. Los dos 1s de la línea superior se combinan con los dos 1s de la fila inferior llevando al término $\bar{B}\bar{C}$. El 1 sobrante, en el cuadrado 0110, se combina con su cuadrado adyacente, 0010, llevando al término $\bar{A}C\bar{D}$. La función optimizada es por tanto

$$F = \bar{B}\bar{D} + \bar{B}\bar{C} + \bar{A}C\bar{D}$$



□ **FIGURA 2-20**
Mapa para el Ejemplo 2-6: $F = \bar{B}\bar{D} + \bar{B}\bar{C} + \bar{A}C\bar{D}$ ■

2-5 MANIPULACIÓN DEL MAPA

Cuando se combinan los cuadrados de un mapa, es necesario asegurar que se incluyen todos los minitérminos de la función. Al mismo tiempo, es necesario minimizar el número de términos de la función optimizada evitando todos los términos redundantes cuyos minitérminos ya están incluidos en otros términos. En esta sección consideramos un procedimiento que ayuda al reconocimiento de patrones útiles en el mapa. Otros temas a tratar son la optimización de productos de sumas y la optimización de funciones incompletas.

Implicantes primos esenciales

El procedimiento para combinar cuadrados en un mapa se podría hacer de forma más sistemática si presentamos los términos «implicante», «implicante primo» e «implicante primo esencial». Un término producto es un *implicante* de una función si la función vale 1 para todos los minitérminos del término producto. Claramente, todos los rectángulos en un mapa compuestos por cuadrados que contienen 1 corresponden a implicantes. Si se elimina cualquier literal de un implicante *P*, resulta un término producto que no es un implicante de la función, entonces *P* es un

implicante primo. En un mapa para una función de *n*-variables, el conjunto de implicantes primos corresponde al conjunto de todos los rectángulos hechos con 2^m cuadrados que contienen 1 ($m = 0, 1, ..., n$), donde cada rectángulo contiene tantos cuadrados como le sea posible.

Si un minitérmino de una función está incluido en un único implicante primo, este implicante primo se llama *esencial*. Así, si un cuadrado que contiene un 1 está en un sólo rectángulo que representa un implicante primo, entonces este implicante primo es esencial. En la Figura 2-15, los términos $\bar{X}Z$ y $X\bar{Z}$ son implicantes primos esenciales, y los términos $X\bar{Y}$ e $\bar{Y}Z$ son implicantes primos no esenciales.

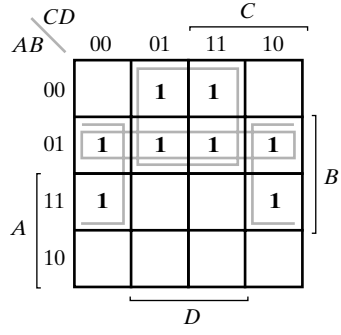
Los implicantes primos de una función se pueden obtener de un mapa de la función como todas las colecciones máximas de 2^m cuadrados que contienen 1 ($m = 0, 1, ..., n$) que constituyen rectángulos. Esto quiere decir que un único 1 en un mapa representa un implicante primo si no es adyacente a cualquier otro 1. Dos 1s adyacentes forman un rectángulo representando un implicante primo, con tal de que no estén dentro de un rectángulo de cuatro o más cuadrados que contengan 1. Cuatro 1s forman un rectángulo que representa un implicante primo si no están dentro de un rectángulo de ocho o más cuadrados que contengan 1, y así sucesivamente. Cada implicante primo esencial contiene al menos un cuadrado que no está dentro de ningún otro implicante primo.

El procedimiento sistemático para encontrar la expresión optimizada del mapa requiere que primero determinemos todos los implicantes primos. Después, se obtiene la expresión optimizada de la suma lógica de todos los implicantes primos esenciales, más otros implicantes primos necesarios para incluir los minitérminos sobrantes que no están incluidos en los implicantes primos esenciales. Este procedimiento se aclarará con ejemplos.

EJEMPLO 2-7 Simplificación usando implicantes primos

Considere el mapa de la Figura 2-21. Hay tres caminos para combinar cuatro cuadrados en rectángulos. Los términos producto obtenidos a partir de estas combinaciones son los implicantes primos de la función, $A\bar{D}$, $B\bar{D}$ y $\bar{A}B$. Los términos $A\bar{D}$ y $B\bar{D}$ son implicantes primos esenciales, pero $\bar{A}B$ no es esencial. Esto es porque los minitérminos 1 y 3 se pueden incluir solamente en el término $\bar{A}D$ y los minitérminos 12 y 14 sólo se puede incluir en el término $B\bar{D}$. Pero los minitérminos 4, 5, 6, y 7 están todos incluidos en dos implicantes primos, uno de ellos es $\bar{A}B$, así el término $\bar{A}B$ no es un implicante primo esencial. De hecho, si se han elegido los implicantes primos esenciales, el tercer término no es necesario porque todos los minitérminos están ya incluidos en los implicantes primos esenciales. La expresión optimizada para la función de la Figura 2-21 es

$$F = \bar{A}D + B\bar{D}$$

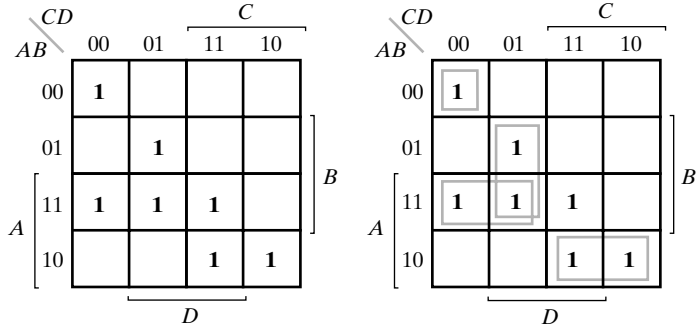


□ **FIGURA 2-21**
Implicantes primos para el Ejemplo 2-7: $\bar{A}D$, $B\bar{D}$, y $A\bar{B}$ ■

EJEMPLO 2-8 Simplificación mediante implicantes primos esenciales y no esenciales

Un segundo ejemplo se muestra en la Figura 2-22. La función dibujada en la parte (a) tiene siete minitérminos. Si intentamos combinar los cuadrados, nos encontramos con seis implicantes primos. Para obtener un número mínimo de términos de la función, tenemos que determinar primero los implicantes primos que son esenciales. Como se muestra en la parte (b) de la figura, la función tiene cuatro implicantes primos esenciales. El término producto $\bar{A}\bar{B}CD$ es esencial porque es el único implicante primo que incluye el minitérmino 0. De igual manera, los términos producto $B\bar{C}D$, $AB\bar{C}$ y $A\bar{B}C$ son implicantes primos esenciales porque son los únicos implicantes primos que incluyen a los minitérminos 5, 12 y 10, respectivamente. El minitérmino 15 está incluido en dos implicantes primos no esenciales. La expresión optimizada para la función consiste en la suma lógica de los cuatro implicantes primos esenciales y un implicante primo que incluye el minitérmino 15:

$$F = \bar{A}\bar{B}\bar{C}\bar{D} + B\bar{C}D + AB\bar{C} + A\bar{B}C + \begin{pmatrix} ACD \\ \text{o} \\ ABD \end{pmatrix}$$



(a) Dibujando los minitérminos (b) Implicantes primos esenciales

□ **FIGURA 2-22**
Simplificación con los implicantes primos del Ejemplo 2-8 ■

La identificación de los implicantes primos esenciales en el mapa proporciona una herramienta adicional que muestra los términos que tienen que aparecer necesariamente en cada expresión de suma de productos de una función y proporciona una estructura parcial para un método más sistemático de elegir patrones de cuadrados.

Implicantes primos no esenciales

Más allá de usar todos los implicantes primos esenciales, se puede aplicar la siguiente regla para incluir los minitérminos restantes de la función en implicantes primos no esenciales:

Regla de selección: minimice el solapamiento entre implicantes primos cuanto sea posible. En particular, en la solución final, asegúrese de que cada implicante primo seleccionado incluye al menos un minitérmino que no está incluido en algún otro implicante primo seleccionado.

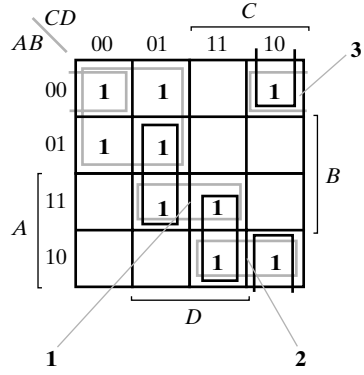
En la mayoría de los casos, esto da lugar a una expresión de suma de producto simplificada, aunque no necesariamente de coste mínimo. El uso de la regla de selección se ilustra en el siguiente ejemplo.

EJEMPLO 2-9 Simplificación de una función usando la regla de selección

Encuentre una forma simplificada de suma de productos para $F(A, B, C, D) = \Sigma m(0, 1, 2, 4, 5, 10, 11, 13, 15)$.

El mapa de F se presenta en la Figura 2-23, mostrando todos los implicantes primos. $\bar{A}\bar{C}$ es el único implicante primo esencial. Usando la anterior regla de selección, podemos elegir los implicantes primos sobrantes para la forma de suma de productos en el orden indicado por los números. Véase como se seleccionan los implicantes primos 1 y 2 para incluir minitérminos sin solapamiento. El implicante primo 3 ($\bar{A}\bar{B}\bar{D}$) y el implicante primo $\bar{B}C\bar{D}$, ambos incluyen el minitérmino 0010 sobrante, el implicante primo 3 se selecciona arbitrariamente para incluir el minitérmino y completar la expresión de suma de productos:

$$F(A, B, C, D) = \bar{A}\bar{C} + ABD + A\bar{B}C + \bar{A}\bar{B}\bar{D}$$



□ **FIGURA 2-23**
Mapa para el Ejemplo 2-9 ■

Optimización de producto de sumas

Las funciones booleanas optimizadas derivadas de los mapas en todos los ejemplos previos han sido expresadas en forma de suma de productos. Con pequeñas modificaciones se puede obtener la forma de producto de sumas.

El procedimiento para obtener una expresión optimizada en forma de producto de suma sale de las propiedades básicas de las funciones booleanas. Los 1s colocados en los cuadrados del mapa representan los minitérminos de la función. Los minitérminos que no están incluidos en la función pertenecen al complemento de la función. De esto vemos que el complemento de una función se representa en el mapa por los cuadrados no marcados con 1. Si marcamos los cuadrados vacíos con 0 y los combinamos en rectángulos válidos, obtenemos una expresión optimizada del complemento de la función. Entonces, tomamos el complemento de $\bar{F}$ para obtener la función F como producto de sumas. Esto se hace tomando el dual y complementando cada literal, como se ha descrito en el Ejemplo 2-2.

EJEMPLO 2-10 Simplificación de una forma de productos de sumas

Simplifique la siguiente función booleana en forma de productos de sumas:

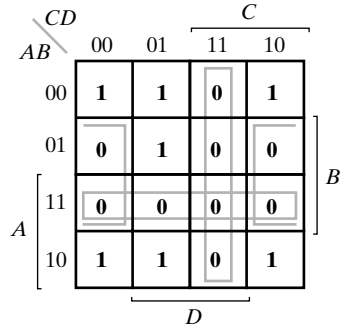
$$F(A, B, C, D) = \Sigma m(0, 1, 2, 5, 8, 9, 10)$$

Los 1s marcados en el mapa de la Figura 2-24 representan los minitérminos de la función. Los cuadrados marcados con 0 representan los minitérminos no incluidos en F y así denotan el complemento de F. Combinando los cuadrados marcados con 0, obtenemos la función complementada optimizada

$$\bar{F} = AB + CD + B\bar{D}$$

Tomando el dual y complementando cada literal se obtiene el complemento de $\bar{F}$. Así es F en forma de producto de sumas:

$$F = (\bar{A} + \bar{B})(\bar{C} + \bar{D})(\bar{B} + D)$$



□ **FIGURA 2-24**
Mapa para el Ejemplo 2-10: $F = (\bar{A} + \bar{B})(\bar{C} + \bar{D})(\bar{B} + D)$ ■

El ejemplo anterior muestra el procedimiento para obtener la optimización en producto de sumas cuando la función originalmente se expresa como suma de minitérminos. El procedimiento también es válido si la función se expresa originalmente como producto de maxitérminos o como producto de sumas. Recuerde que los números del maxitérmino son los mismos que los números del minitérmino de la función complementada, de esta forma se introducen 0s en el mapa para los maxitérminos o para el complemento de la función. Para introducir una función expresada como producto de sumas en el mapa, tomamos el complemento de la función y, de allí, encontramos los cuadrados que hay que marcar con 0. Por ejemplo, se puede dibujar la función

$$F = (\bar{A} + \bar{B} + C)(B + D)$$

en el mapa obteniendo primero el complemento,

$$\bar{F} = AB\bar{C} + \bar{B}\bar{D}$$

y después marcando los 0s en los cuadrados que representan los minitérminos de $\bar{F}$. Los cuadrados sobrantes se marcan con 1. Después, combinando los 1s se llega a la expresión optimizada en forma de suma de productos. Combinando los 0s y después calculando el complemento resulta la expresión optimizada en forma de producto de sumas. Así, para cada función dibujada en el mapa, podemos derivar la función optimizada en cualquiera de las dos formas canónicas.

Condiciones de indiferencia

Los minitérminos de una función booleana especifican todas las combinaciones de valores de variables para los que la función es igual a 1. Se supone que la función es igual a 0 para el resto de los minitérminos. Sin embargo, esta suposición no es siempre válida, ya que hay aplicaciones en las que la función no está especificada para ciertas combinaciones de valores de las variables. Hay dos casos donde ocurre esto. En el primer caso, las combinaciones de entrada no ocurren nunca. Como ejemplo, el código binario de cuatro bits para los dígitos decimales tiene seis combinaciones que no se usan y que no se espera que ocurran. En el segundo caso, las combinaciones de entrada se espera que ocurran, pero el valor de la salida como respuesta a estas combinaciones no importa. En ambos casos se dice que no se han especificado las salidas para estas combinaciones. Las funciones que tienen salidas sin especificar para algunas combinaciones de entradas se llaman *funciones incompletamente especificadas*. En la mayoría de las aplicaciones, simplemente nos da igual qué valor asume la función para los minitérminos no especificados. Por esta razón, es usual llamar los minitérminos no especificados *condiciones de indiferencia*. Estas condiciones se pueden usar en un mapa para proporcionar la función más simplificada.

Habría que darse cuenta de que un minitérmino indiferente no se puede marcar con un 1 en el mapa, porque esto implicaría que la función sería siempre 1 para uno de estos minitérminos. Asimismo, poniendo un 0 en el cuadrado implica que la función es 0. Para distinguir la condición de indiferencia de 1 y 0, se usa una *X*. Así, una *X* dentro de un cuadrado en el mapa indica que no nos importa si está asignado el valor de 0 o 1 a la función para un minitérmino en particular.

Se pueden usar los minitérminos de indiferencia para simplificar la función en un mapa. Cuando se simplifica la función *F*, usando los 1s podemos elegir si incluimos los minitérminos de indiferencia que resultan del implicante primo más simple de *F*. Cuando se simplifica la función $\bar{F}$ usando los 0s, podemos elegir si incluimos los minitérminos de indiferencia que resultan de los implicantes primos más sencillos de $\bar{F}$, independientemente de los que están incluidos en los implicantes primos de *F*. En ambos casos, es irrelevante si los minitérminos de indiferencia están incluidos o no en los términos de la expresión final. El manejo de condiciones de indiferencia se demuestra en el siguiente ejemplo.

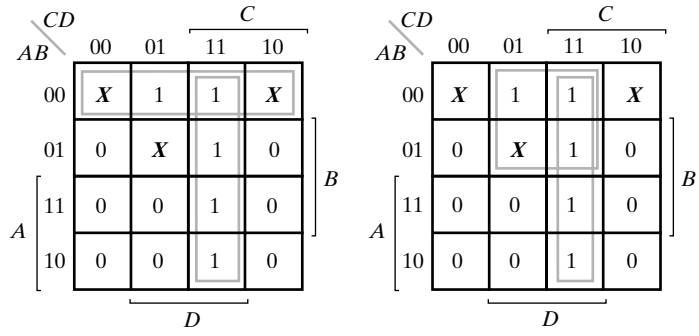
EJEMPLO 2-11 Simplificación con condiciones de indiferencia

Para clarificar el procedimiento en el manejo de condiciones de indiferencia, considere la siguiente función *F* que no está completamente especificada, que tiene tres minitérminos de indiferencia *d*:

$$F(A, B, C, D) = \Sigma m(1, 3, 7, 11, 15)$$

$$d(A, B, C, D) = \Sigma m(0, 2, 5)$$

Los minitérminos de *F* son las combinaciones de variables que igualan la función a 1. Los minitérminos de *d* son minitérminos de indiferencia. La optimización del mapa se muestra en la Figura 2-25. Los minitérminos de *F* están marcados con 1, los de *d* están marcados con *X*, y los cuadrados sobrantes se han rellenado con 0. Para conseguir la función simplificada en forma de suma de productos, tenemos que incluir los cinco 1s en el mapa, pero podemos o no incluir alguna de las *X*, dependiendo de que se produzca la expresión más sencilla de la función. El término *CD* incluye los cuatro minitérminos en la tercera columna. Los minitérminos sobrantes



□ **FIGURA 2-25**
Ejemplo con condiciones de indiferencia

en el cuadrado 0001 se puede combinar con el cuadrado 0011 para dar lugar a un término de tres literales. Sin embargo, incluyendo una o dos *X* adyacentes, podemos combinar cuatro cuadrados en un rectángulo para llegar a un término de dos literales. En la parte (a) de la figura, los minitérminos de indiferencia 0 y 2 están incluidos con los 1s, lo cual da lugar a la función simplificada

$$F = CD + \bar{A}\bar{B}$$

En la parte (b), el minitérmino de indiferencia 5 está incluido con los 1s, y la función simplificada es ahora

$$F = CD + \bar{A}D$$

Las dos expresiones representan dos funciones que son algebraicamente diferentes. Ambas incluyen los minitérminos especificados en la función original incompletamente especificada, pero cada uno incluye diferentes minitérminos de indiferencia. Por lo que respecta a la función incompletamente especificada, ambas expresiones son aceptables. La única diferencia está en el valor de *F* para los minitérminos no especificados.

También es posible obtener una expresión optimizada de producto de sumas para la función de la Figura 2-25. En este caso, la manera de combinar los 0s es incluir los minitérminos de indiferencia 0 y 2 con los 0s, resultando la función optimizada complementada

$$\bar{F} = \bar{D} + A\bar{C}$$

Tomando el complemento de $\bar{F}$ resulta la expresión optimizada en forma de producto de sumas:

$$F = D(\bar{A} + C)$$ ■

El ejemplo anterior muestra que inicialmente se consideran los minitérminos de indiferencia en el mapa representando ambos 0 y 1. El valor 0 o 1 finalmente asignado depende del proceso de optimización. Debido a este proceso, la función optimizada tendrá el valor 0 o 1 para cada minitérmino de la función original, incluyendo los que inicialmente eran indiferentes. Así, aunque las salidas de la especificación inicial podrían contener *X*, las salidas en una implementación particular de la especificación son solamente 0 y 1.

2-6 OPTIMIZACIÓN DE CIRCUITOS MULTINIVEL

Aunque hemos averiguado que la optimización de circuitos de dos niveles puede reducir el coste de los circuitos lógicos combinacionales, muchas veces se puede ahorrar más costes usando circuitos con más de dos niveles. A estos circuitos se les llama circuitos multinivel. Este ahorro se demuestra mediante la implementación de la función:

$$G = ABC + ABD + E + ACF + ADF$$

La Figura 2-26(a) muestra la implementación de dos niveles de G que tiene un coste de 17 entradas de puerta. Ahora suponemos que aplicamos la ley distributiva del Álgebra de Boole a G para conseguir:

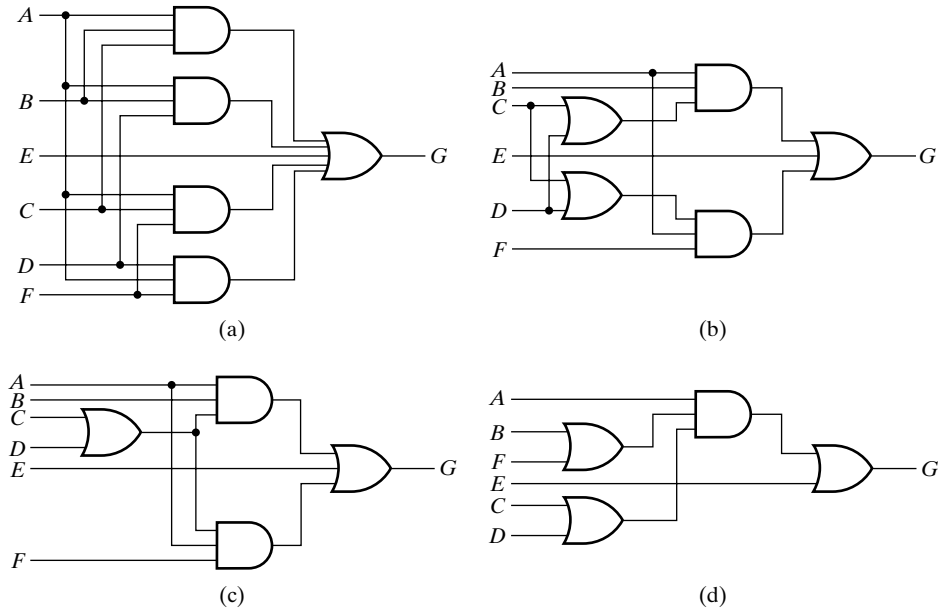
$$G = AB(C + D) + E + A(C + D)F$$

Esta ecuación da lugar a la implementación en varios niveles de G de la Figura 2-26(b) que tiene un coste de 13 entradas de puerta, con una mejora de 4 entradas de puerta. En la Figura 2-26(b), $C + D$ se ha implementado dos veces. En cambio, una implementación de esta subfunción puede compartirse para dar lugar al circuito de la Figura 2-26(c) con un coste de 11 entradas de puerta, con una mejora de 2. Este uso de $(C + D)$ sugiere que se puede escribir G como

$$G = (AB + AF)(C + D) + E$$

Esto incrementa el coste a 12. Pero sacando el factor A de $AB + AF$, obtenemos

$$G = A(B + F)(C + D) + E$$



□ **FIGURA 2-26**
Ejemplo de un circuito multinivel

La Figura 2-26(d) muestra la implementación en varios niveles de *G* usando esta ecuación que tiene un coste de entradas de puerta de solo nueve, que es ligeramente más que la mitad del coste original.

Esta reducción se ha obtenido mediante una secuencia de aplicaciones de identidades algebraicas, observando en cada paso el efecto en el coste de las entradas de puerta. Sólo usando el álgebra de Boole para obtener circuitos simplificados de dos niveles, el procedimiento usado aquí no es realmente sistemático. Además, no existe ningún procedimiento algorítmico para usar los Mapas de Karnaugh para la optimización de circuitos de dos niveles que tengan un coste óptimo, debido al amplio rango de acciones posibles y al número de soluciones posibles. Así la optimización en múltiples niveles se basa en el uso de un conjunto de las transformaciones aplicadas junto con la evaluación del coste para encontrar una solución buena pero no necesariamente óptima. Para el resto de esta sección, tenemos en cuenta estas transformaciones e ilustramos su aplicación para reducir el coste del circuito. Las transformaciones, mostradas en el siguiente ejemplo, se definen como:

1. *Factorización*: es encontrar una forma factorizada de una expresión de suma de productos o de producto de sumas para una función.
2. *Descomposición*: es la expresión de una función como un conjunto de funciones nuevas.
3. *Extracción*: es la expresión de varias funciones como un conjunto de nuevas funciones.
4. *Substitución* de una función *G* por una función *F* es expresar *F* como función de *G* y de algunas o todas las variables originales de *F*.
5. *Eliminación*: es la inversa de la sustitución donde la función *G* dentro de una expresión de función *F* se sustituye por la expresión de *G*. A la eliminación también se le llama *flattening* (aplanar) o *collapsing* (colapsar).

EJEMPLO 2-12 Transformaciones de optimización de múltiples niveles

Se usarán las siguientes funciones para ilustrar las transformaciones:

$$G = A\bar{C}E + A\bar{C}F + A\bar{D}E + A\bar{D}F + BCD\bar{E}\bar{F}$$
$$H = \bar{A}BCD + ACE + ACF + BCE + BCF$$

La primera transformación a mostrar es la factorización usando la función *G*. Inicialmente, miramos a la *factorización algebraica*, que evita axiomas que son únicos en el Álgebra de Boole, tal como los que incluyen el complemento y la idempotencia. Se pueden encontrar los factores no solamente para la expresión entera de *G*, sino también para sus subexpresiones. Por ejemplo, ya que los primeros cuatro términos de *G* contienen la variable *A*, se puede sacar fuera de estos términos dando lugar a en:

$$G = A(\bar{C}E + \bar{C}F + \bar{D}E + \bar{D}F) + BCD\bar{E}\bar{F}$$

En este caso, véase que A y $\bar{C}E + \bar{C}F + \bar{D}E + \bar{D}F$ son factores, y $BCD\bar{E}\bar{F}$ no está involucrado en la operación de factorización. Sacando los factores $\bar{C}$ y $\bar{D}$, $\bar{C}E + \bar{C}F + \bar{D}E + \bar{D}F$ se puede escribir como $\bar{C}(E + F) + \bar{D}(E + F)$ lo que, además, se puede escribir como $(\bar{C} + \bar{D})(E + F)$. La integración de esta expresión en *G* da como resultado:

$$G = A(\bar{C} + \bar{D})(E + F) + BCD\bar{E}\bar{F}$$

El término $BCD\bar{E}\bar{F}$ se podría factorizar en términos producto, pero esta factorización no reduciría el número de entradas de puertas y por tanto no se tiene en cuenta. El número de entradas

por puerta en la expresión original de la suma de productos para G es 26 y en la forma factorizada de G es 18, ahorrándose 8 entradas de puerta. Debido a esta factorización, hay más puertas en serie desde las entradas hasta las salidas, un máximo de cuatro niveles en vez de tres niveles incluyendo los inversores de entrada. Esto daría lugar a un incremento del retardo del circuito después de aplicar un mapeo tecnológico.

La segunda transformación que se muestra es la descomposición que permite operaciones más allá de la factorización algebraica. La forma factorizada de G se puede escribir como una descomposición, según sigue a continuación:

$$G = A(\bar{C} + \bar{D})X_2 + BX_1\bar{E}\bar{F}$$
$$X_1 = CD$$
$$X_2 = E + F$$

Una vez que X_1 y X_2 se han definido, se puede calcular el complemento, y los complementos pueden reemplazar $\bar{C} + \bar{D}$ y $\bar{E}\bar{F}$, respectivamente, en G. Una ilustración de la transformación de sustitución es

$$G = A\bar{X}_1X_2 + BX_1\bar{X}_2$$
$$X_1 = CD$$
$$X_2 = E + F$$

El número de entradas de puertas para esta descomposición es 14, dando lugar a un ahorro de 12 entradas de puerta de la expresión original de suma de productos para G, y de 4 puertas de entrada de la forma factorizada de G.

Para ilustrar la extracción, necesitamos realizar la descomposición en H y extraer subexpresiones comunes en G y H. Sacando el factor B de H, tenemos

$$H = B(\bar{A}CD + AE + A + CE + CF)$$

Determinando factores adicionales en H, podemos escribir

$$H = B(\bar{A}(CD) + (A + C)(E + F))$$

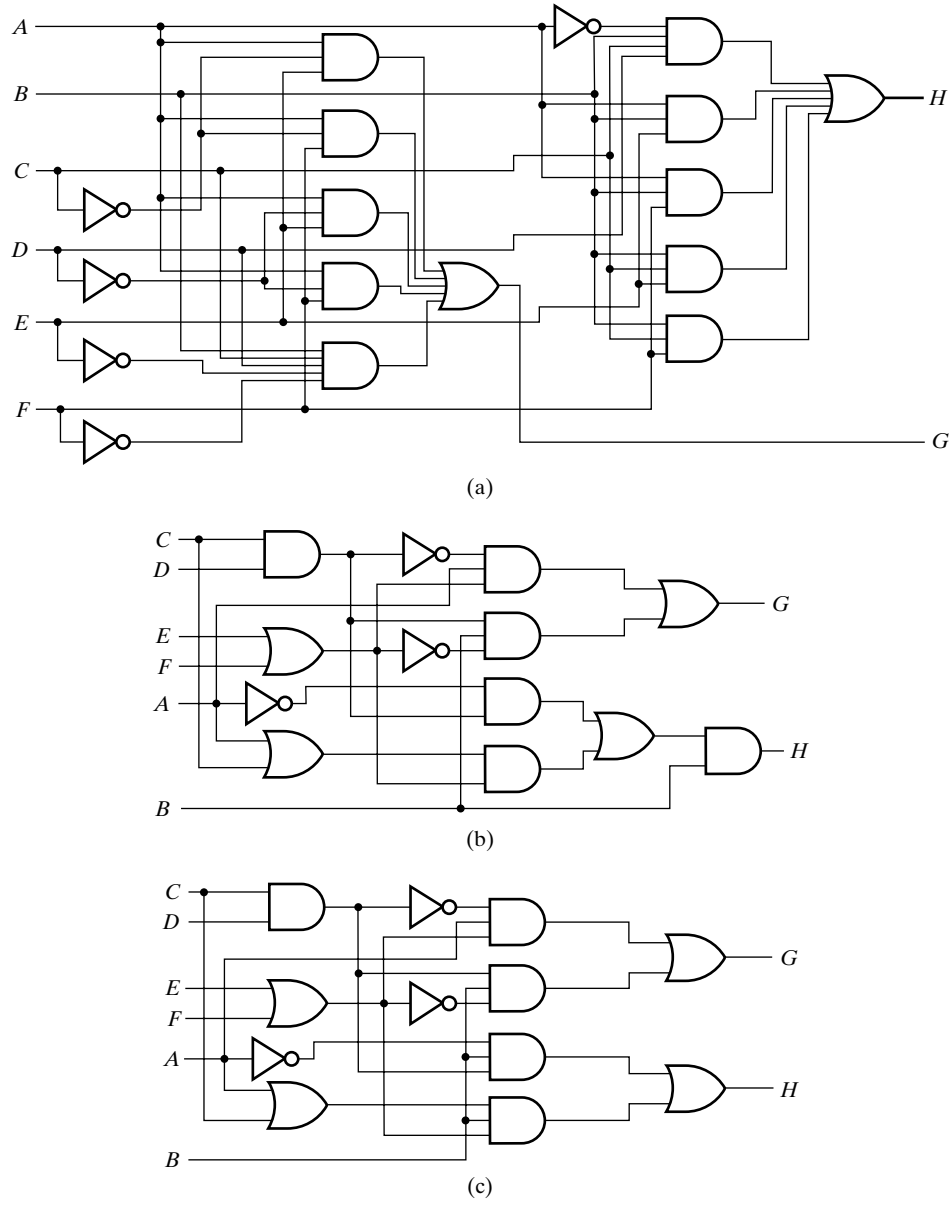
Ahora los factores X_1, X_2, y X_3 se puede extraer para obtener

$$X_1 = CD$$
$$X_2 = E + F$$
$$X_3 = A + C$$

y los factores X_1 y X_2 se puede compartir entre G y H. Realizando la sustitución, podemos escribir G y H como

$$G = A\bar{X}_1X_2 + BX_1\bar{X}_2$$
$$H = B(\bar{A}X_1 + X_3X_2)$$

Se da un diagrama lógico para la suma de productos original en la Figura 2-27(a) y para la forma extraída en la Figura 2-27(b). El coste de entradas de puerta para las funciones originales G y H, sin términos compartidos, excepto para los inversores de entrada, es de 48. Para G y H



□ **FIGURA 2-27**
Ejemplo de optimización de un circuito de múltiples niveles

en forma decompuesta, sin términos compartidos entre *G* y *H*, es de 31. Con términos compartidos es de 25, dividiendo el coste de entradas de puertas de entrada a la mitad. ■

Este ejemplo muestra el valor de las transformaciones reduciendo el coste de entradas. En general, debido al amplio rango de soluciones alternativas y la complejidad de determinar divisores para usar en la descomposición y la extracción, la obtención de soluciones realmente óptimas, en cuanto al coste de entradas, no es factible normalmente, así que solamente se buscan soluciones buenas.

La clave para realizar transformaciones con éxito es la determinación de los factores que se usan en la descomposición o extracción y la elección de la secuencia aplicada en la transformación. Esas decisiones son complejas y fuera del ámbito de nuestros estudios, pero normalmente se incorporan en las herramientas de síntesis lógica.

Nuestra discusión, hasta ahora, trataba solamente de la optimización en varios niveles en términos de reducir el número de entradas de puertas. En muchos diseños, la longitud de la ruta o rutas más largas por el circuito se restringe por el retardo en la ruta, el tiempo que tarda en propagarse el cambio de una señal por un camino a través de las puertas. En estos casos podría ser necesario reducir el número de puertas en serie. Una reducción así, usando la transformación final, eliminación, se muestra en el siguiente ejemplo.

EJEMPLO 2-13 Ejemplo de transformación para la reducción del retardo

En el circuito de la Figura 2-27(b), las rutas de *C, D, E, F* y *A* a *H* pasan por cuatro puertas de 2 entradas. Suponiendo que todas las puertas, con diferente número de entradas, contribuyen con el mismo retardo en la ruta (retardo mayor que el de un inversor) estas son las rutas más lentas del circuito. Debido a una especificación de un retardo de ruta máximo para un circuito, hay que acortar estos caminos en al menos tres puertas de varias entradas o su equivalente en retardos de puertas de varias entradas o de inversores. Este acortamiento de las rutas se debería hacer con un incremento mínimo del número de entradas de puerta.

La transformación de eliminación que reemplaza variables intermediarias, X_i, con las expresiones a su lado derecho o elimina otra factorización como la de la variable *B*, es el mecanismo para reducir el número de puertas en serie. Para determinar qué factor o combinación de factores se debería eliminar, tenemos que contemplar el efecto en el número de entradas de puerta. El incremento en el número de entradas de puertas para las combinaciones de eliminaciones que reducen las longitudes de rutas problemáticas en al menos una puerta de entrada son interesantes. Hay solamente tres de estas combinaciones: eliminación de la factorización de *B*, eliminación de variables intermediarias X_1, X_2, y X_3, y eliminación del factor *B* y las tres variables intermediarias X_1, X_2, y X_3. Los incrementos en los números de entradas de puerta respectivos para estas acciones son 0, 12, y 12, respectivamente. Claramente, la eliminación del factor *B* es la mejor elección ya que el número de entradas de puerta no se incrementa. Esto también demuestra que, debido a la descomposición adicional de *H*, la ganancia del coste de entradas de puerta de 3, que ocurrió sacando el factor *B* al principio, ha desaparecido. El diagrama lógico resultante de la eliminación del factor *B* se muestra en la Figura 2-27(c). ■

Mientras que la reducción necesaria del retardo se ha obtenido usando la eliminación para reducir el número de puertas a lo largo de la ruta en el Ejemplo 2-13, en general, tal reducción de puerta podría no reducir el retardo, o incluso podría incrementarlo debido a las diferencias en las características de retardo de las puertas que se discutirá más adelante en el Capítulo 3.

2-7 OTROS TIPOS DE PUERTAS

Ya que las funciones booleanas se expresan en términos de operaciones AND, OR, y NOT, éstas constituyen un procedimiento directo para implementar una función booleana con puertas AND, OR, y NOT. Sin embargo, encontramos que la posibilidad de considerar puertas con otras operaciones lógicas es de un considerable interés práctico. Los factores que hay que considerar al construir otros tipos de puertas son la viabilidad y economía de implementar la puerta con

componentes electrónicos, la capacidad de la puerta para implementar funciones booleanas por si solas o en combinación con otras puertas, y la utilidad de representar puertas funcionales que se usan frecuentemente. En esta sección introducimos estos otros tipos de puertas que se usan a lo largo del resto del texto. Las técnicas específicas para la incorporación de estos tipos de puertas en los circuitos se muestran en la Sección 3-5.

Los símbolos gráficos y las tablas de verdad de las seis puertas lógicas se muestra en la Figura 2-28, con seis tipos de puertas adicionales mostradas en la Figura 2-29. A las puertas de la Figura 2-28 se las denomina como puertas *primitivas*, y a las de la Figura 2-29 como puertas *complejas*.

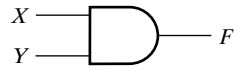
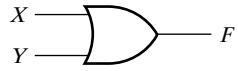
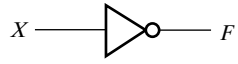
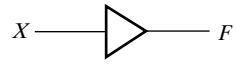
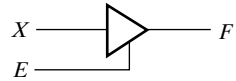
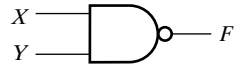
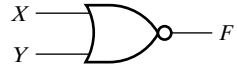
Aunque las puertas de la Figura 2-28 se muestra solamente con dos variables binarias de entrada, *X* e *Y*, y una variable binaria de salida, *F*, con la excepción del inversor y el *buffer*, todos podrían tener más de dos entradas. Las diferentes formas de los símbolos mostrados, tanto como los símbolos rectangulares no mostrados, se especifica en detalle en el «*Institute of Electrical y Electronics Engineers*» (IEEE) *Standard Graphic Symbols for Logic Funcions* (IEEE Standard 91-1984). Las puertas AND, OR, y NOT se han definido anteriormente. El circuito NOT invierte el sentido lógico de una señal binaria para producir una operación de complemento. Recuerde que a este circuito se le llama típicamente *inversor* en vez de puerta NOT. El círculo pequeño a la salida del símbolo gráfico de un inversor se llama formalmente un *indicador de negación* y designa el complemento lógico. Informalmente nos referimos al indicador de negación como a una «burbuja». El mismo símbolo triangular designa un circuito *buffer*. Un *buffer* produce la función lógica $Z = X$, ya que el valor binario de la salida es igual al valor binario de la entrada. Este circuito se usa principalmente para amplificar una señal eléctrica para permitir que más puertas sean conectadas a la salida y se reduce el tiempo de propagación de las señales por el circuito.

El *buffer* triestado es único en el sentido de que se pueden conectar sus salidas entre ellos, con tal de que una sola de las señales de sus entradas *E* sea 1 en un momento dado. Este tipo de *buffer* y su uso básico se discuten en detalle más tarde en esta sección.

La puerta NAND representa el complemento de la operación AND, y la puerta NOR representa el complemento de la operación OR. Los nombres respectivos son abreviaturas de NOT-AND y NOT-OR, respectivamente. Los símbolos gráficos para la puerta NAND y la puerta NOR están compuestos por el símbolo de la puerta AND y el símbolo de la puerta OR, respectivamente, con una burbuja en la salida, que indica el complemento de la operación. En las tecnologías actuales de circuitos integrados, las puertas NAND y NOR son las funciones primitivas naturales para los circuitos más simples y más rápidos. Si consideramos el inversor como una versión degenerada de las puertas NAND y NOR con solamente una entrada, las puertas NAND o puertas NOR, por si mismas, pueden implementar cualquier función booleana. Así, estos tipos de puertas se usan mucho más que las puertas AND y OR en los circuitos lógicos actuales. Como consecuencia, las implementaciones de los circuitos actuales se realiza muchas veces en térinos de este tipo de puertas.

Un tipo de puerta que se puede usar únicamente para implementar todas las funciones booleanas se llama una *puerta universal*. Para mostrar que la puerta NAND es una puerta universal, solo tenemos que demostrar que se pueden obtener las operaciones lógicas de AND, OR, y NOT usando solamente puertas NAND. Esto se ha realizado en la Figura 2-30. La operación complemento obtenida de una puerta NAND con una entrada corresponde a una puerta NOT. De hecho, la puerta NAND de una entrada es un símbolo no válido y se sustituye por un símbolo NOT, como se muestra en la figura. La operación AND requiere una puerta NAND seguida de una puerta NOT. El NOT invierte la salida de la NAND, resultando una operación AND. La operación OR se logra usando una puerta NAND con un NOT en cada entrada. Si se aplica el

Símbolos gráficos

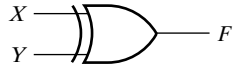
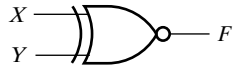
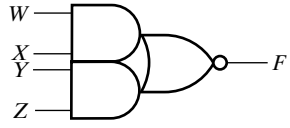
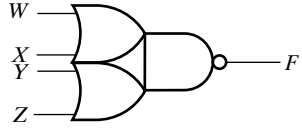
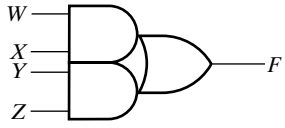
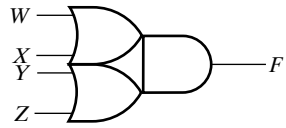
Nombre	Símbolo	Ecuación algebraica	Tabla de verdad
AND		$F = XY$	$X\ Y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 1$
OR		$F = X + Y$	$X\ Y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 1$
NOT (inversor)		$F = \overline{X}$	$X \mid F$ $0 \mid 1$ $1 \mid 0$
Buffer		$F = X$	$X \mid F$ $0 \mid 0$ $1 \mid 1$
Buffer triestado			$E\ X \mid F$ $0\ 0 \mid$ Hi-Z $0\ 1 \mid$ Hi-Z $1\ 0 \mid 0$ $1\ 1 \mid 1$
NAND		$F = \overline{X \cdot Y}$	$X\ Y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$
NOR		$F = \overline{X + Y}$	$X\ Y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 0$

□ **FIGURA 2-28**
Puertas lógicas digitales primitivas

Teorema de DeMorgan, como se muestra en la Figura 2-30, las inversiones se anulan y resulta una función OR.

La puerta OR exclusiva (XOR) mostrada en la Figura 2-29 es similar a la puerta OR, pero excluye (tiene el valor 0 para) la combinación con ambas entradas X e Y iguales a 1. El símbolo gráfico para la puerta XOR es similar al de la puerta OR, excepto en la línea curvada adicional en las entradas. La OR exclusiva tiene el símbolo especial $\oplus$ para designar esta operación. La

Símbolos gráficos

Nombre	Símbolo	Ecuación algebraica	Tabla de verdad
OR exclusiva (XOR)		$F = X\overline{Y} + \overline{X}Y$ $= X \oplus Y$	$X\ Y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$
NOR exclusiva (XNOR)		$F = XY + \overline{X}\overline{Y}$ $= \overline{X \oplus Y}$	$X\ Y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 1$
AND-OR-INVERSOR (AOI)		$F = \overline{WX + YZ}$	
OR-AND -INVERSOR (OAI)		$F = \overline{(W + X)(Y + Z)}$	
AND-OR (AO)		$F = WX + YZ$	
OR-AND (OA)		$F = (W + X)(Y + Z)$	

□ **FIGURA 2-29**
Puertas lógicas digitales complejas primitivas

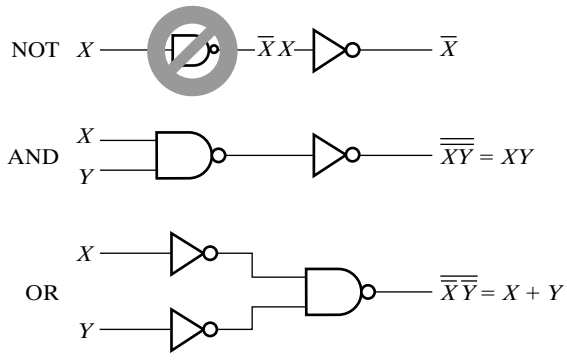
NOR exclusiva es el complemento de la OR exclusiva, como se indica con la burbuja a la salida de su símbolo gráfico.

La puerta AND-OR-INVERT (AOI) forma el complemento de una suma de productos. Hay muchas puertas diferentes AND-OR-INVERT dependiendo del número de puertas AND y el número de entradas en cada AND y sus salidas van conectadas directamente a la puerta OR. Por ejemplo, suponga que la función implementada por un AOI es

$$F = \overline{XY + Z}$$

A esta AOI se le denomina como una AOI 2-1 ya que consiste en una AND de 2 entradas y una entrada directa a la puerta OR. Si la función implementada es

$$F = \overline{TUV + WX + YZ}$$



□ **FIGURA 2-30**
Operaciones lógicas con puertas NAND

entonces a la AOI se le llama AOI 3-2-2. LA OR-AND-INVERT (OAI) es la dual de la AOI e implementa el complemento en forma de producto de sumas. La AND-OR (AO) y OR-AND (OA) son versiones de AOI y OAI sin el complemento.

En general, las puertas complejas se usan para reducir la complejidad del circuito necesario para la implementación de funciones específicas de Boole con el fin de reducir costes del circuito integrado. Además, reducen el tiempo necesario para la propagación de señales por el circuito.

CIRCUITOS CMOS Este suplemento, que discute la implementación de puertas primitivas y complejas en tecnología CMOS, está disponible en la página web del texto: www.librosite.net/Mano.

2-8 OPERADOR Y PUERTAS OR EXCLUSIVA

Además de la puerta de OR exclusiva mostrada en la Figura 2-29, hay un operador de OR exclusivo con sus identidades algebraicas propias. El operador OR exclusivo, denotado por $\oplus$, es una operación lógica que ejecuta la función

$$X \oplus Y = X\overline{Y} + \overline{X}Y$$

Es igual a 1 si sólo una variable de entrada es igual a 1. El operador NOR exclusivo, también conocido como *equivalencia*, es el complemento del OR exclusivo y se expresa mediante la función

$$\overline{X \oplus Y} = XY + \overline{X}\,\overline{Y}$$

Es igual a 1 si ambas entradas, X e Y, son iguales a 1 o si ambas entradas son iguales a 0. Se puede demostrar que las dos funciones son complementos una de la otra, tanto por medio de la tabla de verdad o, como sigue a continuación, por manipulación algebraica:

$$\overline{X \oplus Y} = \overline{X\overline{Y} + \overline{X}Y} = (\overline{X} + Y)(X + \overline{Y}) = XY + \overline{X}\,\overline{Y}$$

Las siguientes identidades se pueden aplicar a la operación OR exclusiva:

$$X \oplus 0 = X \qquad X \oplus 1 = \overline{X}$$
$$X \oplus X = 0 \qquad X \oplus \overline{X} = 1$$
$$X \oplus \overline{Y} = \overline{X \oplus Y} \qquad \overline{X} \oplus Y = \overline{X \oplus Y}$$

Se puede verificar cada una de estas identidades usando una tabla de verdad o reemplazando la operación $\oplus$ por su expresión booleana equivalente. También se puede mostrar que la operación OR exclusiva es tanto conmutativa como asociativa; o sea,

$$A \oplus B = B \oplus A$$

$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

Esto significa que las dos entradas a una puerta de OR exclusiva pueden ser intercambiadas sin tener efecto en la operación. También significa que podemos evaluar una operación OR exclusiva de tres variables en cualquier orden y por esa razón, se pueden expresar las ORs exclusivas con tres o más variables sin paréntesis.

Una función OR exclusiva de dos entradas se puede construir con puertas convencionales. Se usan dos puertas NOT, dos puertas AND, y una puerta OR. La asociatividad del operador OR exclusivo sugiere la posibilidad de puertas de OR exclusivas con más que dos entradas. Sin embargo, el concepto del OR exclusivo para más de dos variables se reemplaza por la función impar discutida a continuación. Por esto, no hay ningún símbolo para la OR exclusiva de más de dos entradas. Por dualidad, se reemplaza la NOR exclusiva por la función par y no tiene ningún símbolo para más de dos entradas.

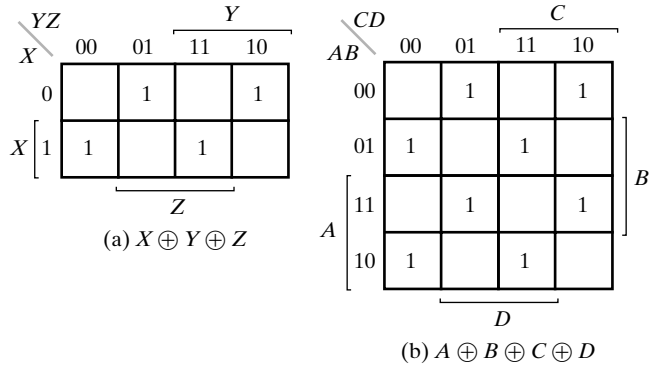
Función impar

La operación OR exclusiva con tres o más variables se puede convertir en una función booleana ordinaria reemplazando el símbolo $\oplus$ con su expresión booleana equivalente. En concreto, el caso de tres variables puede convertirse en una expresión booleana como la siguiente:

$$\begin{aligned} X \oplus Y \oplus Z &= (X\bar{Y} + \bar{X}Y)\bar{Z} + (XY + \bar{X}\bar{Y})Z \\ &= X\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + \bar{X}\bar{Y}Z + XYZ \end{aligned}$$

La expresión booleana indica claramente que la OR exclusiva de tres variables es igual a 1 si solamente una variable es igual a 1 o si las tres variables son iguales a 1. Por esto, mientras que en la función de dos variables sólo una variable tiene que ser igual a 1, con tres o más variables tiene que ser un número impar de variables iguales a 1. Como consecuencia, se define la operación OR exclusiva de múltiples variables como *función impar*. De hecho, estrictamente hablando, este es el nombre correcto para la operación $\oplus$ con tres o más variables; el nombre «OR exclusiva» es aplicable al caso con solamente dos variables.

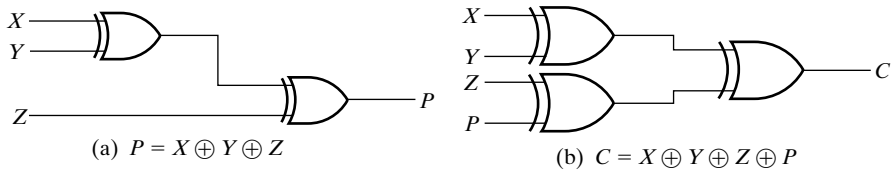
La definición de la función impar se puede clarificar dibujando la función en un mapa. La Figura 2-31(a) muestra el mapa para la función impar de tres variables. Los cuatro minitérminos de la función son diferentes entre si en al menos dos literales y por esto no pueden estar adyacentes en el mapa. Se dice que estos minitérminos tienen una *distancia* de dos uno al otro. La función impar se identifica por los cuatro minitérminos cuyas variables binarias tienen un número impar de 1. El caso de cuatro variables se muestra en la Figura 2-31(b). Los ocho minitérminos marcados con 1 en el mapa constituyen la función impar. Véase el patrón característico de la distancia entre los 1 en el mapa. Se debería mencionar que los minitérminos no marcados con 1 en el mapa tienen un número impar de 1 y constituyen el complemento de la función impar, llamada *función par*. La función impar se implementa mediante puertas OR exclusiva de dos entradas, como se muestra en la Figura 2-32. La función par se puede obtener reemplazando la puerta de salida con una puerta NOR exclusiva.



(a) $X \oplus Y \oplus Z$

(b) $A \oplus B \oplus C \oplus D$

□ **FIGURA 2-31**
Mapas para funciones impares de múltiples variables



(a) $P = X \oplus Y \oplus Z$

(b) $C = X \oplus Y \oplus Z \oplus P$

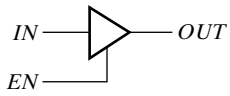
□ **FIGURA 2-32**
Funciones impares de múltiples entradas

2-9 SALIDAS EN ALTA IMPEDANCIA

Hasta ahora, hemos considerado solamente puertas que tienen los valores de salida 0 lógico y 1 lógico. En esta sección introducimos dos estructuras importantes, los *buffers* triestado y las puertas de transmisión, que proporcionan un tercer valor de salida a la que se llama como *estado de alta impedancia* y que se denota como Hi-*Z* o simplemente *Z* o *z*. El valor Hi-*Z* se comporta como un circuito abierto, que quiere decir que, mirando hacia atrás del circuito, encontramos que la salida aparece como desconectada. Las salidas de alta impedancia podrían aparecer en cualquier puerta, pero aquí nos restringimos a dos estructuras de puertas con entradas de datos simples. Las puertas con valores de salida Hi-*Z* pueden tener sus salidas conectadas entre sí, con tal de que no pueda haber dos puertas que conduzcan al mismo tiempo con valores opuestos 0 y 1. Por contra, las puertas con salidas lógicas de 0 y 1 no pueden tener sus salidas conectadas.

Buffers triestado El *buffer* triestado se ha presentado anteriormente como una de las puertas primitivas. Como indica su nombre, una salida lógica de tres estados muestra tres estados diferentes. Dos de los estados son el 1 y el 0 lógico de la lógica convencional. El tercer estado es el valor *Hi-Z*, al cual, para la lógica triestado, se le denomina como *estado Hi-Z* o estado de alta impedancia.

El símbolo gráfico y la tabla de verdad para un *buffer* triestado se presenta en la Figura 2-33. El símbolo de la Figura 2-33(a) se distingue del símbolo de un *buffer* normal por su entrada de habilitación, *EN*, que entra por debajo del símbolo. Según la tabla de verdad de la



(a) Símbolo lógico

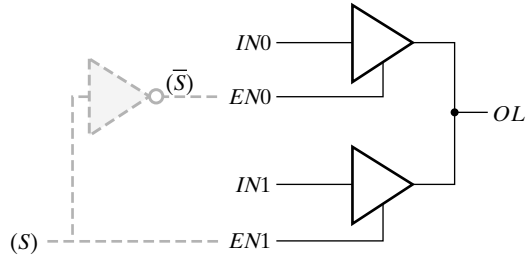
EN	*IN*	*OUT*
0	*X*	*Hi-Z*
1	0	0
1	1	1

(b) Tabla de verdad

□ **FIGURA 2-33**
Buffer de tres estados

Figura 2-33(b), si $EN = 1$, *OUT* es igual a *IN*, comportándose como un buffer normal. Pero para $EN = 0$, el valor de salida es de alta impedancia (*Hi-Z*), independiente del valor de *IN*.

Las salidas del buffer triestado se pueden conectar para formar una salida multiplexada. La Figura 2-34(a) muestra dos buffers triestados con sus salidas conectadas para formar la salida *OL*. Centramos el estudio en la salida de esta estructura en función de las cuatro entradas *EN*1, *EN*0, *IN*1, y *IN*0. El comportamiento de la salida se muestra en la tabla de verdad de la Figura 2-34(b). Para *EN*1 y *EN*0 igual a 0, ambas salidas del buffer son *Hi-Z*. Ya que ambas aparecen como circuitos abiertos, *OL* también es un circuito abierto, representado por un valor *Hi-Z*. Para $EN1 = 0$ y $EN0 = 1$, la salida del buffer superior es *IN*0 y la salida del buffer inferior es *Hi-Z*. Ya que el valor de *IN*0 combinado con un circuito abierto es justamente *IN*0, *OL* tiene el valor *IN*0, dando lugar a la segunda y tercera fila de la tabla de verdad. El caso contrario ocurre para $EN1 = 1$ y $EN0 = 0$, así *OL* tiene el valor *IN*1, dando lugar a la cuarta y quinta fila de la tabla



(a) Diagrama lógico

*EN*1	*EN*0	*IN*1	*IN*0	*OL*
0	0	*X*	*X*	*Hi-Z*
(*S*) 0	**($\overline{S}$)** 1	*X*	0	0
0	1	*X*	1	1
1	0	0	*X*	0
1	0	1	*X*	1
1	1	0	0	0
1	1	1	1	1
1	1	0	1	
1	1	1	0	

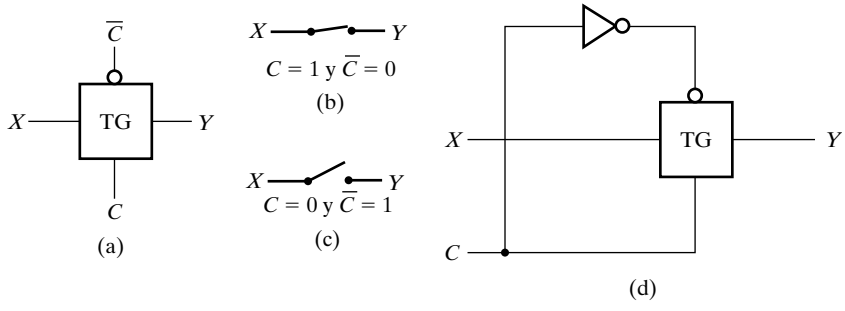
(b) Tabla de verdad

□ **FIGURA 2-34**
Buffers de tres estados formando una línea multiplexada OL

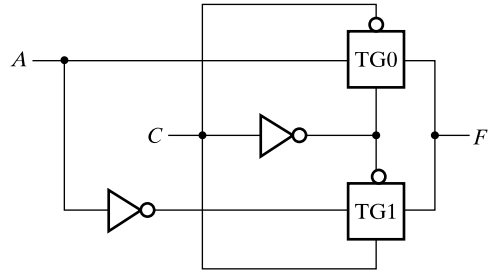
de verdad. Para *EN*1 y *EN*0, ambas 1, la situación es más complicada. Si *IN*1 = *IN*0, su valor aparece en *OL*. Pero si *IN* ≠ *IN*0, sus valores tienen un conflicto en la salida. El conflicto resulta en un flujo de corriente de la salida del buffer que está en 1 hacia la salida del buffer que está en 0. Esta corriente muchas veces es lo suficientemente alta para producir calentamiento y incluso podría destruir el circuito, como se simboliza por los iconos de «humo» en la tabla de verdad. Claramente hay que evitar semejante situación. El diseñador tiene que asegurar que *EN*0 y *EN*1 no son nunca iguales a 1 al mismo tiempo. En el caso general, para buffers triestados vinculados a una línea de bus, *EN* puede ser igual a 1 para solamente uno de los buffers y tiene que ser 0 para el resto. Una posibilidad para asegurar esto es usar un decodificador para generar las señales de *EN*. Para el caso de dos buffers, el decodificador es solamente un inversor con entrada seleccionable *S*, como se muestra en las líneas punteadas de la Figura 2-34(a). Es interesante examinar la tabla de verdad con el inversor puesto. Observe el área sombreada de la tabla de la Figura 2-34(b). Claramente, el valor en *S* selecciona entre las entradas *IN*0 y *IN*1. Además, la salida del circuito OL no está nunca en el estado Hi-*Z*.

Puertas de transmisión En los circuitos integrados lógicos, hay un circuito lógico con transistores CMOS que es suficientemente importante para ser presentado por separado, a nivel de puertas. Este circuito, llamado puerta de transmisión (*transmission gate* (TG)), es una especie de interruptor electrónico para conectar y desconectar dos puntos en un circuito. La Figura 2-35(a) muestra el símbolo IEEE para la puerta de transmisión. Tiene cuatro conexiones externas o puertos. *C* y $\overline{C}$ son las entradas de control y *X* e *Y* son las señales que van a ser conectadas o desconectadas por la TG. En la Figura 2-35(*b*) y (*c*), aparece el modelo con interruptores para la puerta de transmisión. Si $C = 1$ y $\overline{C} = 0$, *X* e *Y* se conectan según el modelo, por un interruptor «cerrado» y las señales pueden pasar de *X* a *Y* o de *Y* a *X*. Si $C = 0$ y $\overline{C} = 1$, *X* e *Y* están desconectados, como se representa en el modelo, por un interruptor «abierto» y las señales no pueden pasar entre *X* e *Y*. En uso normal, las entradas de control están conectadas por un inverso, como se muestra en la Figura 2-35(d), de manera que *C* y $\overline{C}$ son uno el complemento del otro.

Para ilustrar el uso de una puerta de transmisión, se muestra en la Figura 2-36(a) una puerta OR exclusiva construida con dos puertas de transmisión y dos inversores. La entrada *C* controla las rutas por las puertas de transmisión, y la entrada *A* proporciona la salida para *F*. Si la entrada *C* es igual a 1, existe un camino por la puerta de transmisión TG1 conectando *F* con $\overline{A}$, y no existe ninguna ruta por TG0. Si la entrada *C* es igual a 0, existe una ruta por TG0 conectando *F* con *A*, y no existe ninguna ruta por TG1. Así, la salida *F* está conectada con *A*. Esto da lugar a la tabla de verdad de la OR exclusiva, como se indica en la Figura 2-36(b).



□ **FIGURA 2-35**
Puerta de transmisión (TG)



(a)

A	*C*	TG1	TG0	*F*
0	0	Abierta	Cerrada	0
0	1	Cerrada	Abierta	1
1	0	Abierta	Cerrada	1
1	1	Cerrada	Abierta	0

(b)

□ **FIGURA 2-36**
Puerta de transmisión OR exclusiva

2-10 RESUMEN DEL CAPÍTULO

Las operaciones lógicas primitivas AND, OR, y NOT definen tres componentes lógicos llamados puertas, usadas para implementar sistemas digitales. El Álgebra de Boole, definido en términos de estas operaciones, proporciona una herramienta para manipular funciones booleanas en el diseño de circuitos lógicos. Las formas canónicas de minitérminos y maxitérminos se corresponden directamente con las tablas de verdad de las funciones. Estas formas canónicas se pueden manipular en forma de suma de productos o producto de sumas, que corresponden a circuitos con dos niveles puertas. Dos medidas de coste para minimizar, optimizando un circuito, son el número de literales de entrada al circuito y el número total de puertas del circuito. Los mapas-K, desde dos hasta cuatro variables, son una alternativa efectiva a la manipulación algebraica en la optimización de circuitos pequeños. Estos mapas se pueden usar para optimizar las formas de suma de productos, producto de sumas, y funciones especificadas incompletamente con condiciones de indiferencia. Se han ilustrado las transformaciones para optimizar circuitos de múltiples niveles con tres o más niveles de puertas.

Las operaciones primitivas AND y OR no están directamente implementadas por elementos lógicos primitivos en la familia lógica más popular. Así, se han presentado las primitivas NAND y NOR tanto así como las puertas complejas que implementan esas familias. Se ha presentado una primitiva más compleja, la OR exclusiva, tanto como su complemento, la NOR exclusiva, junto con sus propiedades matemáticas.

REFERENCIAS

1. BOOLE, G.: *An Investigation of the Laws of Thought*. New York: Dover, 1854.
2. KARNAUGH, M.: *«A Map Method for Synthesis of Combinational Logic Circuitos», Transactions of AIEE, Communication y Electronics*, 72, part I (Nov. 1953), 593-99.
3. DIETMEYER, D. L.: *Logic Design of Digital Systems*, 3rd ed. Boston: Allyn Bacon, 1988.
4. MANO, M. M.: *Digital Design*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2002.
5. ROTH, C. H.: *Fundamentals of Logic Design*, 4th ed. St. Paul: West, 1992.
6. HAYES, J. P.: *Introduction to Digital Logic Design*. Reading, MA: Addison-Wesley, 1993.
7. WAKERLY, J. F.: *Digital Design*: *Principles y Practices*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2000.
8. GAJSKI, D. D.: *Principles of Digital Design*. Upper Saddle River, NJ: Prentice Hall, 1997.

9. *IEEE Standard Graphic Symbols for Logic Funcións*. (Includes IEEE Std 91a-1991 Supplement y IEEE Std 91-1984.) New York: The Institute of Electrical y Electronics Engineers, 1991.

PROBLEMAS

El signo (+) indica problemas más avanzados y el asterisco (*) indica que hay una solución disponible en la dirección de Internet: http://www.librosite.net/Mano.

2-1. *Demuestre por medio de tablas de verdad la validez de las siguientes identidades:

(a) Teorema de DeMorgan para tres variables: $\overline{XYZ} = \bar{X} + \bar{Y} + \bar{Z}$
(b) La segunda ley distributiva: $X + YZ = (X + Y)(X + Z)$
(c) $\bar{X}Y + \bar{Y}Z + X\bar{Z} = X\bar{Y} + Y\bar{Z} + \bar{X}Z$

2-2. *Demuestre la identidad de cada una de las siguientes ecuaciones booleanas, usando la manipulación algebraica:

(a) $\bar{X}\bar{Y} + \bar{X}Y + XY = \bar{X} + Y$
(b) $\bar{A}B + \bar{B}\bar{C} + AB + \bar{B}C = 1$
(c) $Y + \bar{X}Z + X\bar{Y} = X + Y + Z$
(d) $\bar{X}\bar{Y} + \bar{Y}Z + XZ + XY + Y\bar{Z} = \bar{X}\bar{Y} + XZ + Y\bar{Z}$

2-3. +Demuestre la identidad de cada una de las siguientes ecuaciones booleanas, usando la manipulación algebraica:

(a) $AB + B\bar{C}\bar{D} + \bar{A}BC + \bar{C}D = B + \bar{C}D$
(b) $WY + \bar{W}YZ + WXZ + \bar{W}X\bar{Y} = WY + \bar{W}X\bar{Z} + \bar{X}Y\bar{Z} + X\bar{Y}Z$
(c) $A\bar{C} + \bar{A}B + \bar{B}C + \bar{D} = (\bar{A} + \bar{B} + \bar{C} + \bar{D})(A + B + C + \bar{D})$

2-4. +Dado que $A \cdot B = 0$ y $A + B = 1$, use la manipulación algebraica para demostrar que

$$(A + C) \cdot (\bar{A} + B) \cdot (B + C) = B \cdot C$$

2-5. +En este capítulo se ha usado un Álgebra específica de Boole con solamente dos elementos 0 y 1. Se pueden definir otras álgebras booleanas con más que dos elementos usando elementos que corresponden a cadenas binarias. Estas álgebras forman la base matemática para las operaciones lógicas de bit a bit que vamos a estudiar en el Capítulo 7. Suponga que las cadenas son cada una un nibble (medio byte) de cuatro bits. Entonces hay 2^4, o 16, elementos en el álgebra, donde un elemento *I* es el nibble de 4-bit en binario correspondiente a *I* en decimal. Basándose en la aplicación bit a bit del Álgebra de Boole de dos elementos, defina cada uno de los siguientes puntos para la nueva álgebra de manera que las identidades booleanas sean correctas:

(a) La operación OR $A + B$ para cada dos elementos *A y B*
(b) La operación AND $A \cdot B$ para cada dos elementos *A y B*
(c) El elemento que actúa como el 0 para el álgebra
(d) El elemento que actúa como el 1 para el álgebra
(e) Para cada elemento *A*, el elemento $\bar{A}$.

2-6. Simplifique las siguientes expresiones booleanas a las expresiones conteniendo un número mínimo de literales:

(a) $\bar{A}\bar{C} + \bar{A}BC + \bar{B}C$

(b) $\overline{(A + B)}(\bar{A} + \bar{B})$
(c) $ABC + \bar{A}C$
(d) $BC + B(AD + \bar{C}D)$
(e) $(B + \bar{C} + B\bar{C})(BC + A\bar{B} + AC)$

2-7. *Reduzca las siguientes expresiones booleanas al número de literales indicado:
(a) $\bar{X}\bar{Y} + XYZ + \bar{X}Y$ a tres literales
(b) $X + Y(Z + \overline{X + Z})$ a dos literales
(c) $\bar{W}X(\bar{Z} + \bar{Y}Z) + X(W + \bar{W}YZ)$ a un literal
(d) $(AB + \bar{A}\bar{B})(\bar{C}\bar{D} + CD) + \bar{A}\bar{C}$ a cuatro literales

2-8. Usando el Teorema de DeMorgan, exprese la función

$$F = \bar{A}BC + \bar{B}\bar{C} + A\bar{B}$$

(a) Solamente con operaciones de OR y de complemento.
(b) Solamente con operaciones AND y de complemento.

2-9. *Encuentre el complemento de las siguientes expresiones:
(a) $A\bar{B} + \bar{A}B$
(b) $(\bar{V}W + X)Y + \bar{Z}$
(c) $WX(\bar{Y}Z + Y\bar{Z}) + \bar{W}\bar{X}(\bar{Y} + Z)(Y + \bar{Z})$
(d) $(A + \bar{B} + C)(\bar{A}\bar{B} + C)(A + \bar{B}\bar{C})$

2-10. *Obtenga la tabla de verdad para las siguientes funciones, y exprese cada función en forma de suma de minitérminos y de producto de maxitérminos:
(a) $(XY + Z)(Y + XZ)$
(b) $(\bar{A} + B)(\bar{B} + C)$
(c) $WX\bar{Y} + WX\bar{Z} + WXZ + Y\bar{Z}$

2-11. Para las Funciones de Boole E y F, según la siguiente tabla de verdad:

X	*Y*	*Z*	*E*	*F*
0	0	0	1	0
0	0	1	0	0
0	1	0	1	1
0	1	1	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	1

(a) Enumere los minitérminos y maxitérminos de cada función.
(b) Enumere los minitérminos de $\bar{E}$ y $\bar{F}$.
(c) Enumere los minitérminos de $E + F$ y $E \cdot F$.
(d) Exprese E y F en forma algebraica de suma de minitérminos.
(e) Simplifique E y F a expresiones con un número mínimo de literales.

2-12. *Convierta las siguientes expresiones en formas de suma de productos y de producto de sumas:

(a) $(AB + C)(B + \bar{C}D)$
(b) $\bar{X} + X(X + \bar{Y})(Y + \bar{Z})$
(c) $(A + B\bar{C} + CD)(\bar{B} + EF)$

2-13. Dibuje el diagrama lógico para las siguientes expresiones booleanas. El diagrama debería corresponder exactamente a la ecuación. Suponga que los complementos de las entradas no están disponibles.

(a) $W\bar{X}\bar{Y} + \bar{W}Z + YZ$
(b) $A(B\bar{D} + \bar{B}D) + D(BC + \bar{B}\bar{C})$
(c) $W\bar{Y}(X + Z) + \bar{X}Z(W + Y) + W\bar{X}(Y + Z)$

2-14. Optimice las siguientes ecuaciones booleanas mediante un mapa de tres variables:

(a) $F(X, Y, Z) = \Sigma m(1, 3, 6, 7)$
(b) $F(X, Y, Z) = \Sigma m(3, 5, 6, 7)$
(c) $F(A, B, C) = \Sigma m(0, 1, 2, 4, 6)$
(d) $F(A, B, C) = \Sigma m(0, 3, 4, 5, 7)$

2-15. *Optimice la siguiente expresión booleana usando un mapa:

(a) $\bar{X}\bar{Z} + Y\bar{Z} + XYZ$
(b) $\bar{A}B + \bar{B}C + \bar{A}\bar{B}\bar{C}$
(c) $\bar{A}\bar{B} + A\bar{C} + \bar{B}C + \bar{A}B\bar{C}$

2-16. Optimice las siguientes funciones booleanas mediante un mapa de cuatro variables:

(a) $F(A, B, C, D) = \Sigma m(2, 3, 8, 9, 10, 12, 13, 14)$
(b) $F(W, X, Y, Z) = \Sigma m(0, 2, 5, 6, 8, 10, 13, 14, 15)$
(c) $F(A, B, C, D) = \Sigma m(0, 2, 3, 7, 8, 10, 12, 13)$

2-17. Optimice las siguientes funciones booleanas, usando un mapa:

(a) $F(W, X, Y, Z) = \Sigma m(0, 2, 5, 8, 9, 11, 12, 13)$
(b) $F(A, B, C, D) = \Sigma m(3, 4, 6, 7, 9, 12, 13, 14, 15)$

2-18. *Encuentre los minitérminos de las siguientes expresiones dibujando primero cada expresión en un mapa:

(a) $XY + XZ + \bar{X}YZ$
(b) $XZ + \bar{W}X\bar{Y} + WXY + \bar{W}YZ + W\bar{Y}Z$
(c) $\bar{B}\bar{D} + ABD + \bar{A}BC$

2-19. *Encuentre todos los implicantes primos para las siguientes funciones booleanas, y determine cuáles son esenciales:

(a) $F(W, X, Y, Z) = \Sigma m(0, 2, 5, 7, 8, 10, 12, 13, 14, 15)$
(b) $F(A, B, C, D) = \Sigma m(0, 2, 3, 5, 7, 8, 10, 11, 14, 15)$
(c) $F(A, B, C, D) = \Sigma m(1, 3, 4, 5, 9, 10, 11, 12, 13, 14, 15)$

2-20. Optimice las siguientes funciones booleanas encontrando todos los implicantes primos y los implicantes primos esenciales aplicando la regla de selección:

(a) $F(W, X, Y, Z) = \Sigma m(0, 1, 4, 5, 7, 8, 9, 12, 14, 15)$

(b) $F(A, B, C, D) = \Sigma m(1, 5, 6, 7, 11, 12, 13, 15)$
(c) $F(W, X, Y, Z) = \Sigma m(0, 2, 3, 4, 5, 7, 8, 10, 11, 12, 13, 15)$

2-21. Optimice las siguientes funciones booleanas en forma de producto de sumas:

(a) $F(W, X, Y, Z) = \Sigma m(0, 2, 3, 4, 8, 10, 11, 15)$
(b) $F(A, B, C, D) = \Pi M(0, 2, 4, 5, 8, 10, 11, 12, 13, 14)$

2-22. Optimice las siguientes expresiones en (1) forma de suma de productos y (2) forma de producto de sumas:

(a) $A\bar{C} + \bar{B}D + \bar{A}CD + ABCD$
(b) $(\bar{A} + \bar{B} + \bar{D})(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{D})(B + \bar{C} + \bar{D})$
(c) $(\bar{A} + \bar{B} + D)(\bar{A} + \bar{D})(A + B + \bar{D})(A + \bar{B} + C + D)$

2-23. Optimice las siguientes funciones en forma de (1) suma de productos y (2) forma de producto de sumas:

(a) $F(A, B, C, D) = \Sigma m(2, 3, 5, 7, 8, 10, 12, 13)$
(b) $F(W, X, Y, Z) = \Pi M(2, 10, 13)$

2-24. Optimice las siguientes funciones booleanas F junto con las condiciones de indiferencia d:

(a) $F(A, B, C, D) = \Sigma m(0, 3, 5, 7, 11, 13)$, $d(A, B, C, D) = \Sigma m(4, 6, 14, 15)$
(b) $F(W, X, Y, Z) = \Sigma m(0, 6, 8, 13, 14)$, $d(W, X, Y, Z) = \Sigma m(2, 4, 7, 10, 12)$
(c) $F(A, B, C) = \Sigma m(0, 1, 2, 4, 5)$, $d(A, B, C) = \Sigma m(3, 6, 7)$

2-25. *Optimice las siguientes funciones booleanas F junto con las condiciones de indiferencia d. Encuentre todos los implicantes primos y los implicantes primos esenciales, y aplique la regla de selección.

(a) $F(A, B, C) = \Sigma m(3, 5, 6)$, $d(A, B, C) = \Sigma m(0, 7)$
(b) $F(W, X, Y, Z) = \Sigma m(0, 2, 4, 5, 8, 14, 15)$, $d(W, X, Y, Z) = \Sigma m(7, 10, 13)$
(c) $F(A, B, C, D) = \Sigma m(4, 6, 7, 8, 12, 15)$, $d(A, B, C, D) = \Sigma m(2, 3, 5, 10, 11, 14)$

2-26. Optimice las siguientes funciones booleanas F junto con las condiciones de indiferencia d en forma de (1) suma de productos y (2) producto de sumas:

(a) $F(A, B, C, D) = \Pi M(1, 3, 4, 6, 9, 11)$, $d(A, B, C, D) = \Sigma m(0, 2, 5, 10, 12, 14)$
(b) $F(W, X, Y, Z) = \Sigma m(3, 4, 9, 15)$, $d(W, X, Y, Z) = \Sigma m(0, 2, 5, 10, 12, 14)$

2-27. Use la descomposición para encontrar el menor número de entradas de puerta, implementaciones de múltiples niveles, para las funciones usando puertas AND y OR y inversores.

(a) $F(A, B, C, D) = A\bar{B}C + \bar{A}BC + A\bar{B}D + \bar{A}BD$
(b) $F(W, X, Y, Z) = WY + XY + \bar{W}XZ + W\bar{X}Z$

2-28. Use extracción para encontrar el menor número de entradas de puertas compartidas, implementación de múltiple nivel para el par de funciones dadas usando puertas AND y OR y inversores.

(a) $F(A, B, C, D) = \Sigma m(0, 5, 11, 14, 15)$, $d(A, B, C, D) = \Sigma m(10)$
(b) $G(A, B, C, D) = \Sigma m(2, 7, 10, 11, 14)$, $d(A, B, C, D) = \Sigma m(15)$

2-29. Use eliminación para aplanar (flatten) cada uno de los conjuntos de funciones dados en una forma de suma de productos de dos niveles.

(a) $F(A, B, G, H) = AB\bar{G} + \bar{B}G + \bar{A}\bar{H}$, $G(C, D) = C\bar{D} + \bar{C}D$, $H(B, C, D) = B + CD$

(b) $T(U, V, Y, Z) = YZU + \bar{Y}\bar{Z}V$, $U(W, X) = W + \bar{X}$, $V(W, X, Y) = \bar{W}Y + X$

2-30. *Demuestre que el dual del OR exclusiva es igual a su complemento.

2-31. Implemente la siguiente función booleana con puertas de OR exclusiva y AND, usando el menor número de entradas de puerta:

$$F(A, B, C, D) = AB\bar{C}D + A\bar{D} + \bar{A}D$$

2-32. **(a)** Implemente la función $H = \bar{X}Y + XZ$ usando dos *buffers* triestados y un inversor.

(b) Construya una puerta de OR exclusiva interconectando dos *buffers* triestados y dos inversores.

2-33. **(a)** Conecte las salidas de tres *buffers* triestados, y añada la lógica adicional para implementar la función

$$F = \bar{A}BC + ABD + A\bar{B}\bar{D}$$

Suponga que C, D y $\bar{D}$ son entradas de datos a los buffers y A y B pasan por una lógica que genera las entradas de habilitación.

(b) ¿Está su diseño de la parte (a) libre de conflictos en la salida triestado? Si no, cambie el diseño si es necesario para estar libre de estos conflictos.

2-34. Use solamente puertas de transmisión e inversores para implementar la función del Problema 2-32.

2-35. Dependiendo del diseño y de la familia lógica usada, en general no es una buena idea dejar la salida de un circuito triestado o de puertas de transmisión en un estado de alta impedancia (Hi-Z).

(a) Para el circuito de puertas de transmisión diseñado en el Problema 2-33, presente todas las combinaciones de entrada para las que la salida F está en un estado de alta impedancia.

(b) Modifique la lógica de habilitación cambiando las entradas de habilitación de manera que la salida sea 0 o 1 (en vez de Hi-Z).

CAPÍTULO

3

DISEÑO LÓGICO COMBINACIONAL

En este capítulo, aprenderemos a diseñar circuitos combinacionales. Se introducirá el uso de la jerarquía y del diseño *top-down*, ambos esenciales para el diseño de circuitos digitales. Además se realizará una breve descripción del diseño asistido por computadora, incluyendo los lenguajes de descripción *hardware* (HDLs) y la síntesis lógica, dos conceptos que juegan papeles cruciales en el diseño eficiente de los modernos, y complejos, circuitos.

En la sección del espacio de diseño se cubren los conceptos relativos a la tecnología subyacente para la implementación de circuitos digitales. Se presentan las propiedades de las puertas lógicas, los niveles de integración, y los parámetros de las distintas tecnologías lógicas. Se definirán los términos *fan-in*, *fan-out* y tiempo de propagación de las puertas, y se introducirán los conceptos de lógica positiva y negativa. Finalmente, trataremos sobre el equilibrio entre las distintas dimensiones del espacio de diseño, como pueden ser coste y prestaciones.

Se presenta un ciclo de diseño con cinco pasos principales. Los tres primeros pasos, especificación, formulación y optimización se muestran con ejemplos. Se presentan las tecnologías no programables y con ellas el siguiente proceso del ciclo de diseño, el mapeado tecnológico. El paso final en el proceso de diseño, la verificación, se muestra mediante un ejemplo usando tanto el método manual como la simulación lógica. El capítulo finaliza con una introducción a las tecnologías basadas en lógica programable.

Los distintos conceptos de este capitulo son fundamentales en el diseño de una computadora genérica tal y como se mostró en el diagrama del principio del Capítulo 1. Los conceptos vistos en este capítulo se aplican a todos los componentes digitales de esta computadora genérica incluyendo las memorias.

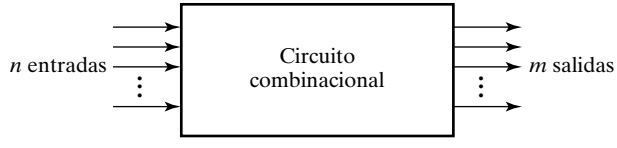
3-1 CONCEPTOS DE DISEÑO Y AUTOMATIZACIÓN

En al Capítulo 1 aprendimos sobre números binarios y códigos binarios capaces de representar cantidades discretas de información. En el Capítulo 2, se presentaron las distintas puertas lógicas y se aprendió a optimizar las expresiones y ecuaciones booleanas con el fin de minimizar el numero de puertas en las implementaciónes. El propósito de este capitulo es aprovechar los conocimientos adquiridos en los capítulos anteriores para formular un procedimiento de diseño sistemático de circuitos combinacionales. Además, los pasos de este proceso se relacionarán con el uso de las herramientas de diseño asistido por computadora. Varios ejemplos servirán para adquirir práctica en el diseño. El diseño digital moderno hace uso de una serie de técnicas y herramientas que son esenciales para el diseño de circuitos complejos y sistemas. El diseño jerárquico, el diseño *top-down*, las herramientas de diseño asistido por computadora, los lenguajes de descripción *hardware*, y la síntesis lógica, son algunas de las herramientas mas importantes necesarias para un diseño digital eficiente y efectivo.

Los circuitos lógicos para sistemas digitales pueden ser combinacionales o secuenciales. Un circuito combinacional está formado por puertas lógicas cuyas salidas están determinadas, en cualquier instante del tiempo, por operaciones lógicas realizadas sobre las entradas en ese mismo instante. Un circuito combinacional realiza una operación lógica que puede especificarse lógicamente mediante un conjunto de ecuaciones booleanas. Por el contrario, los circuitos secuenciales emplean elementos que almacenan bits. Las salidas de un circuito secuencial son función de las entradas y de los bits almacenados en los elementos de memoria que, a su vez, no sólo dependen de los valores actuales de sus entradas sino también de sus valores pasados. El comportamiento de un circuito se debe especificar por una secuencia temporal de entradas así como de bits memorizados. Los circuitos secuenciales serán estudiados en el Capítulo 6.

Un circuito combinacional consiste en variables de entrada, variables de salida, puertas lógicas e interconexiones. Las puertas lógicas interconectadas aceptan señales procedentes de las entradas y generan señales hacia las salidas. En la Figura 3-1, se muestra el diagrama de bloques de un circuito combinacional típico. Las n variables de entrada provienen del entorno del circuito, y las m variables de salida están disponibles para ser usadas por dicho entorno. Cada variable de entrada y de salida existe físicamente como una señal binaria que toma el valor 0 lógico o 1 lógico.

Para n variables de entrada, existen 2^n posibles combinaciones binarias de entrada. Para cada combinación binaria de las variables de entrada, existe un único valor binario posible en cada salida. De esta forma, un circuito combinacional puede definirse a través de una tabla de verdad que muestre una lista con los valores de la salida para cada combinación de las variables de entrada. Un circuito combinacional, puede definirse también mediante m funciones booleanas, una para cada variable de salida. Cada una de estas funciones se expresa mediante una función de las n variables de entrada. Antes de entrar a definir el proceso de diseño, se presentarán dos conceptos fundamentales relacionados con él: el diseño jerárquico y el diseño *top-down*.



□ **FIGURA 3-1**
Diagrama de bloques de un circuito combinacional

Diseño jerárquico

Un circuito se puede representar mediante un símbolo que muestra sus entradas y sus salidas y una descripción que defina exactamente su modo de funcionamiento. Sin embargo, en términos de implementación, un circuito se compone de puertas lógicas y otras estructuras lógicas interconectadas entre sí. Un sistema digital complejo puede llegar a contener millones de puertas lógicas interconectadas. De hecho, un sólo procesador de muy alta escala de integración (VLSI) puede llegar a contener varias decenas de millones de puertas. Con tal complejidad, la interconexión de puertas parece un incomprensible laberinto. De esta manera, no es posible diseñar circuitos o sistemas complejos mediante la simple interconexión de puertas. Con el fin de poder manejar circuitos de semejante complejidad se emplea el método «divide y vencerás». El circuito se divide en piezas menores que llamaremos *bloques*. Los bloques se interconectan para formar el circuito. Las funciones de estos bloques y sus interfaces se definen cuidadosamente de modo que el circuito formado mediante su interconexión obedezca las especificaciones del circuito primigenio. Si un bloque todavía resultase demasiado grande y complejo para diseñarse como una entidad simple, puede volverse a dividir en otros bloques más pequeños. Este proceso puede repetirse tantas veces como sea necesario. Tenga presente que aunque ahora estamos trabajando fundamentalmente con circuitos lógicos, se ha empleado el termino «circuito» para esta discusión pero, estas ideas pueden aplicarse igualmente a los «sistemas» tratados en los siguientes capítulos.

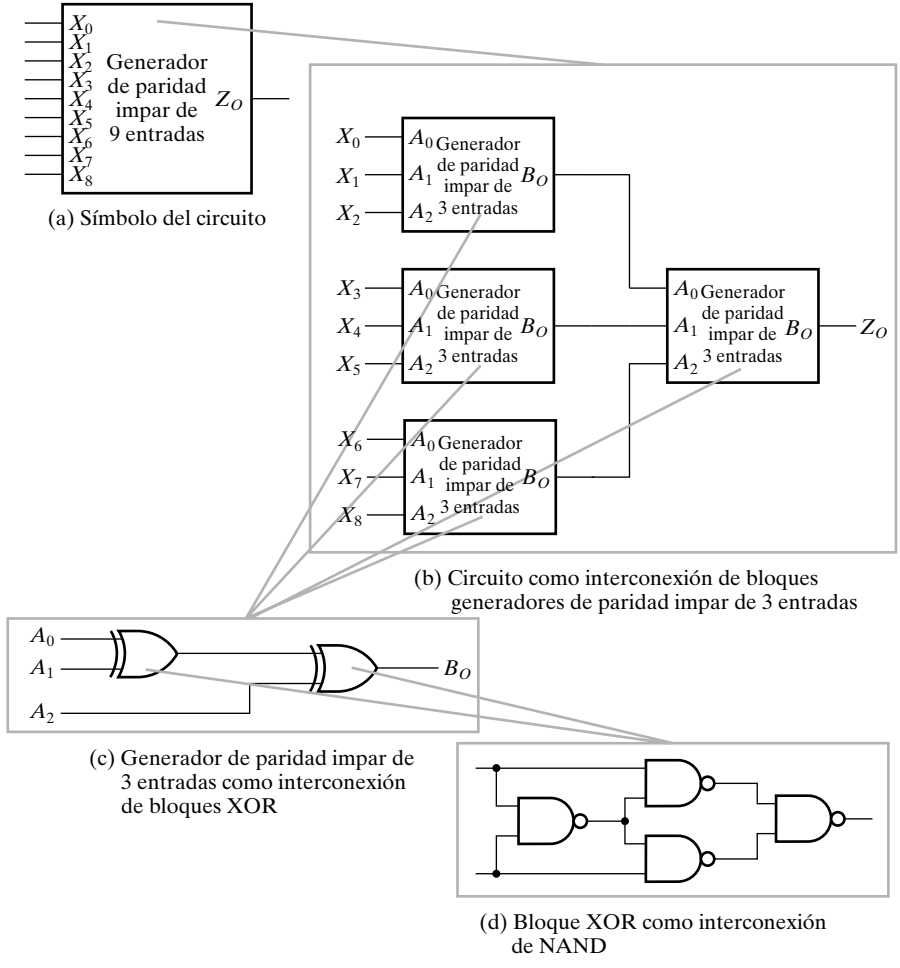
La técnica «divide y vencerás» se muestra para un circuito combinacional en la Figura 3-2. El circuito implementa la función de paridad impar de 9 entradas. La parte (a) de la figura muestra un símbolo para el circuito completo. En la parte (b), se representa un diagrama lógico, esquema o esquemático, para el circuito representado por el símbolo de la parte (a). En este esquemático el diseñador ha decidido partir el circuito en cuatro bloques idénticos, cada uno de los cuales es una función de paridad impar de 3 entradas. Por ello el símbolo empleado para esta función de paridad impar de 3 entradas se repite 4 veces. Los cuatro símbolos se interconectan para formar el circuito de paridad impar de 9 entradas. En la parte (c) se muestra cómo el bloque generador de paridad impar de 3 entradas esta compuesto a su vez por dos puertas XOR interconectadas. Finalmente, en la parte (d), las puertas XOR se implementan a partir de puertas NAND. Tenga en cuenta que en cada caso, cada vez que se desciende un nivel, cada símbolo se sustituye por un esquema que representa la implementación de dicho símbolo.

Este modo de entender el diseño se conoce como *diseño jerárquico* y la relación de símbolos y esquemas resultante constituye una *jerarquía* que representa al circuito diseñado. La estructura de la jerarquía se puede representar, sin las interconexiones, comenzando por el bloque superior y enlazándolo por debajo a aquellos que lo forman. Usando esta representación, la jerarquía para el circuito de paridad impar de 9 entradas se muestra en la Figura 3-3(a). Observe cómo la estructura que resulta tiene forma de árbol invertido. Las «hojas» del árbol son las puertas NAND, en este caso 32 puertas. Con el fin de obtener una representación mas compacta, podemos reutilizar los bloques tal y como se muestra en la Figura 3-3 (b). Este diagrama se corresponde con los bloques de la Figura 3-2 pero empleando solo una copia de cada uno de ellos. Este diagrama y los circuitos de la Figura 3-2 serán de ayuda para ilustrar algunos conceptos útiles asociados con las jerarquías y con los bloques jerárquicos.

En primer lugar, la jerarquía reduce la complejidad necesaria para representar el esquemático de un circuito. Por ejemplo, en la Figura 3-3(a), aparecen 32 bloques NAND. Esto significa que si un generador de paridad impar de 9 entradas va a ser diseñado directamente en términos de puertas NAND, el esquema del circuito consistirá en 32 símbolos de puertas NAND interconectados entre sí, frente a los sólo 10 símbolos necesarios para describir la implementación

jerárquica del circuito que aparece en la Figura 3-2. De esta forma, la jerarquía permite representaciones simplificadas de circuitos complejos.

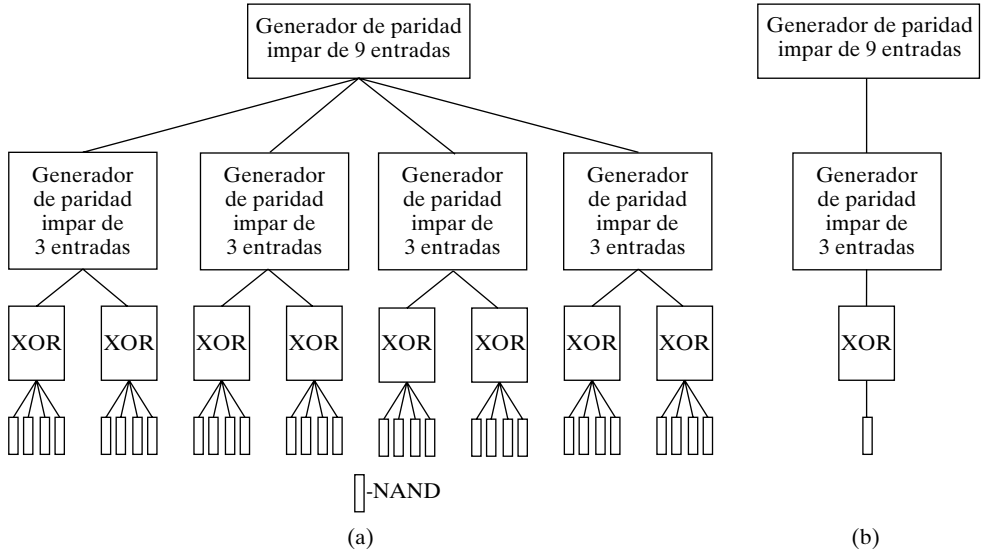
En segundo lugar, la jerarquía de la Figura 3-3 termina en un conjunto de «hojas». En este caso, las hojas son las puertas NAND. Ya que las puertas NAND son circuitos electrónicos, y aquí solo estamos interesados en diseño lógico, las puertas NAND se denominan *bloques primitivos*, *primitivas de diseño* o, simplemente, *primitivas*. Estos son los bloques básicos que teniendo símbolo no tiene esquema lógico. Las primitivas son un tipo rudimentario de *bloques predefinidos*. En general, otras estructuras más complejas que igualmente tienen símbolo pero no esquema lógico, también son bloques predefinidos. En vez de esquemáticos, su función puede definirse mediante un programa o una descripción que sirva como modelo de funcionamiento. Por ejemplo, en la jerarquía representada en la Figura 3-3, las puertas XOR pueden considerarse como bloques predefinidos. En tal caso, el diagrama que describe los bloques XOR de la Figura 3-2 (d) no sería necesario. La representación jerárquica de la Figura 3-3 terminaría entonces en los bloques XOR. En cualquier jerarquía las hojas son los bloques predefinidos, algunos de los cuales pueden ser primitivas.



□ **FIGURA 3-2**
Ejemplo de diseño jerárquico y reutilización de bloques

Una tercera propiedad muy importante que resulta del diseño jerárquico es la reutilización de bloques, tal y como se muestra en la Figuras 3-3(a) y (b). En la parte (a), hay cuatro copias del bloque de paridad impar de 3 entradas y ocho copias del bloque de XOR. En la parte (b), sólo hay una copia del bloque de paridad impar de 3 entradas y una única copia del bloque XOR. Esto significa que el diseñador solo tiene que diseñar un bloque para el generador de paridad impar de 3 entradas y un bloque para la XOR y luego podrá usar dichos bloques, cuatro y ocho veces respectivamente para formar el circuito de paridad impar de 9 entradas. En general, suponga que en varios niveles de la jerarquía los bloques usados han sido definidos cuidadosamente de manera que muchos sean idénticos. Para estos bloques repetidos, solo será necesario un único diseño. Este diseño podrá usarse en todos los lugares donde se necesite el bloque. Cada presencia de un determinado bloque en el diseño se denomina una *instancia* del bloque y su uso se llama *instanciación*. Un bloque es *reutilizable* en el sentido que se puede emplear en múltiples lugares del diseño del circuito y, posiblemente, también en el diseño de otros circuitos. Este concepto reduce enormemente los esfuerzos necesarios para el diseño de circuitos complejos. Véase que, en la implementación del circuito, cada instancia de un bloque requiere su propio *hardware* como se muestra en la Figura 3-3(a). La reutilización, como se presenta en la Figura 3-3(b), únicamente tiene cabida en el esquema, no en la implementación real del circuito.

Tras completar la discusión acerca del proceso de diseño, en los Capítulos 4 y 5, centraremos nuestra atención en los bloques reutilizables predefinidos que usualmente están situados en los niveles inferiores de la jerarquía de los diseños lógicos. Son bloques que proporcionan funciones básicas empleadas en diseño digital. Permiten a los diseñadores trabajar, la mayor parte del tiempo, en el ciclo de diseño a un nivel superior al de las primitivas. Nos referiremos a estos bloques particulares como *bloques funcionales*. De este modo, un bloque funcional es una colección predefinida de puertas interconectadas. Muchos de estos bloques funcionales han estado disponibles durante décadas como circuitos de integración a media escala (MSI) que se interconectaban para formar circuitos o sistemas. Actualmente, las librerías de las herramientas



□ **FIGURA 3-3**
Diagramas representando la jerarquía de la Figura 3-2

de diseño asistido por computadora incluyen bloques similares que se emplean para el diseño de cicuitos integrados mas grandes. Estos bloques funcionales forman un catalogo de componentes digitales elementales que se usan ampliamente en el diseño y la implementación de circuitos integrados para computadoras y sistemas digitales.

Diseño *top-down*

Preferiblemente, el proceso de diseño debe ser realizado de arriba hacia abajo (*top-down*). Esto significa que el funcionamiento del circuito se especifica mediante texto o mediante un lenguaje de descripción *hardware* (HDL), junto con los objetivos de coste, prestaciones y fiabilidad. En los niveles altos del diseño el circuito se divide en bloques, tantas veces como sea necesario, hasta conseguir bloques suficientemente pequeños como para permitir su diseño lógico. Para un diseño lógico manual, puede ser necesario dividir todavía más los bloques. En síntesis automática, la descripción HDL se convierte automáticamente en lógica. Entonces, tanto para el diseño manual como para la síntesis automática, la lógica se optimiza y es entonces *mapeada* a las primitivas disponibles. En la práctica, la realidad se separa significativamente de esta descripción ideal, especialmente en los niveles más altos del diseño. Para alcanzar la reusabilidad y para maximizar el uso de módulos predefinidos es a menudo necesario realizar algunas partes del diseño de abajo hacia arriba (*bottom-up*). Además, un diseño concreto obtenido durante el ciclo de diseño puede violar alguno de los objetivos de la especificación inicial. En estos casos es necesario recorrer hacia arriba la jerarquía de diseño hasta alcanzar algún nivel en el que esta violación pueda ser eliminada. Entonces se revisa a ese nivel una parte del diseño y las revisiones son transferidas desde ese nivel hacia abajo a través de la jerarquía.

En este texto, dado que la familiaridad del lector con el diseño lógico y de computadoras es, probablemente, limitada; es necesario disponer ya de un conjunto de bloques funcionales que permitan guiar el diseño *top-down*. De la misma forma, también debe ser alcanzada una cierta maestría en la manera cómo dividir un circuito en bloques que puedan servir como guía para un diseño *top-down*. Así, una gran parte de este texto se dedicará al diseño *bottom-up* más que al *top-down*. Para comenzar construyendo las bases del diseño *top-down*, en los Capítulo 4 y 5 centraremos nuestros esfuerzos en el diseño de los bloques funcionales de uso más frecuente. En los Capítulos 7 y 10 ilustraremos cómo circuitos más grandes y sistemas, pueden ser divididos en bloques y cómo estos bloques son implementados con bloques funcionales. Finalmente, comenzando en el Capítulo 11, aplicaremos estas ideas para contemplar el diseño más desde una perspectiva *top-down*.

Diseño asistido por computadora

El diseño de sistemas complejos y de circuitos integrados no resultaría factible sin la ayuda de herramientas de diseño asistido por computadora (CAD). Las herramientas de captura de esquemas permiten dibujar bloques e interconectarlos en cualquiera de los niveles de la jerarquía. En el nivel de las primitivas y los bloques funcionales, se proporcionan *librerías* de símbolos gráficos. Las herramientas de captura de esquemas ayudan a construir una jerarquía permitiendo generar símbolos de bloques jerárquicos y replicando estos símbolos para poder reutilizarlos.

Las primitivas y los bloques funcionales de las librerías tienen asociados unos modelos que permiten verificar el comportamiento y los tiempos característicos tanto de los bloques jerárquicos como del circuito entero. Esta verificación se realiza aplicando entradas a los bloques o al circuito, y usando un *simulador lógico* que determine las salidas. Mostraremos esta simulación lógica con algunos ejemplos.

Las primitivas de las librerías también tienen asociada otra información como es área física o parámetros de retardo, que puede ser empleada por el *sintetizador lógico* para optimizar diseños generados automáticamente a partir de una especificación en un lenguaje de descripción *hardware*.

Lenguajes de descripción *hardware*

Hasta ahora, hemos mencionado los lenguajes de descripción *hardware* sólo ocasionalmente. Sin embargo, en el diseño moderno estos lenguajes han llegado a ser cruciales en el ciclo de diseño. Inicialmente, justificaremos el empleo de dichos lenguajes describiendo sus aplicaciones. Más tarde, discutiremos brevemente sobre VHDL y Verilog®, los más populares de ellos. Comenzando en el Capítulo 4, se presentarán con detalle ambos lenguajes, aunque esperamos que en cualquier otro curso que se imparta sólo se vea uno de ellos.

Los lenguajes de descripción *hardware* son parecidos a los lenguajes de programación, pero están más orientados a describir estructuras *hardware* y sus comportamientos. Se distinguen fundamentalmente de los lenguajes de programación en que describen operaciones en paralelo mientras que la mayoría de los lenguajes de programación describen operaciones secuenciales. Una aplicación obvia para los lenguajes de descripción *hardware* es proporcionar una alternativa a los esquemáticos. Cuando el lenguaje se emplea de esta manera para describir una interconexión de componentes, se denomina *descripción estructural*. La descripción estructural, referida como *netlist*, se emplea como entrada en la simulación lógica tal y como se haría con el esquemático. Para poder llevar a cabo esto, es necesario disponer de modelos para cada primitiva. Si se emplea un HDL, entonces estos modelos pueden también escribirse en dicho HDL, con lo que se logra una representación más uniforme y portable para la simulación.

Sin embargo, la potencia del HDL se hace más evidente cuando se emplea para algo más que para representar información esquemática. Puede representar ecuaciones booleanas, tablas de verdad y operaciones complejas como las aritméticas. De esta manera, en el diseño *top-down*, se puede definir con exactitud la descripción, a muy alto nivel, de un sistema entero empleando HDL. Cómo una fase del ciclo de diseño, esta descripción de alto nivel puede redefinirse y subdividirse en descripciones de más bajo nivel. Finalmente, y como resultado del ciclo de diseño, se obtiene una descripción final en términos de primitivas y bloques funcionales. Tenga presente que todas estas descripciones pueden simularse. Ya que todas estas descripciones representan el mismo sistema en términos de funciones, pero no necesariamente de tiempo, siempre deben responder entregando los mismos valores lógicos para las mismas entradas. Esta vital propiedad de la simulación permite la verificación del diseño y es una de las causas principales que justifican el empleo de HDLs.

Otra de las causas responsables del amplio uso de los HDLs es la síntesis lógica. La descripción HDL de un sistema puede definirse a un nivel intermedio denominado nivel de transferencia entre registros (RTL). Una herramienta de síntesis lógica acompañada de una librería de componentes puede convertir dicha descripción en una interconexión de primitivas que implementa el circuito. De esta forma, al eliminar el diseño lógico manual se consigue que el diseño de lógica compleja sea mucho más eficiente.

Actualmente existen dos HDL: VHDL y Verilog® que son muy empleados como lenguajes estándar para el diseño *hardware*. Los lenguajes estándar son definidos, aprobados y publicados por el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE). Cualquier implementación de estos lenguajes debe cumplir sus respectivos estándares. Este proceso de estandarización da a los HDLs una ventaja más sobre los esquemáticos. Los HDLs son portables a través de distintas herramientas de diseño asistido por computadora mientras que las herramientas de captura

de esquemas suelen ser únicas para cada fabricante. Además, de estos lenguajes estándar un buen numero de compañías disponen de sus propios lenguajes, a veces desarrollados mucho antes que los lenguajes estándar, y que incorporan características únicas para sus productos particulares.

VHDL significa lenguaje de descripción *hardware* para VHSIC. VHDL fue desarrollado bajo un contrato del Departamento de Defensa de los Estados Unidos como parte del programa de circuitos integrados de muy alta velocidad (VHSIC) para convertirse, posteriormente, en un estándar del IEEE. Verilog® fue desarrollado por una compañía, Gateway Design Automation, más tarde adquirida por Cadence® Design Systems, Inc. Durante un tiempo, Verilog® fue un lenguaje propietario pero posteriormente se convirtió en un lenguaje estándar del IEEE. En este texto haremos una breve introducción a ambos VHDL y Verilog®. Estas partes del texto son opcionales y permitirán a su profesor abordar cualquiera de los lenguajes o ninguno de ellos.

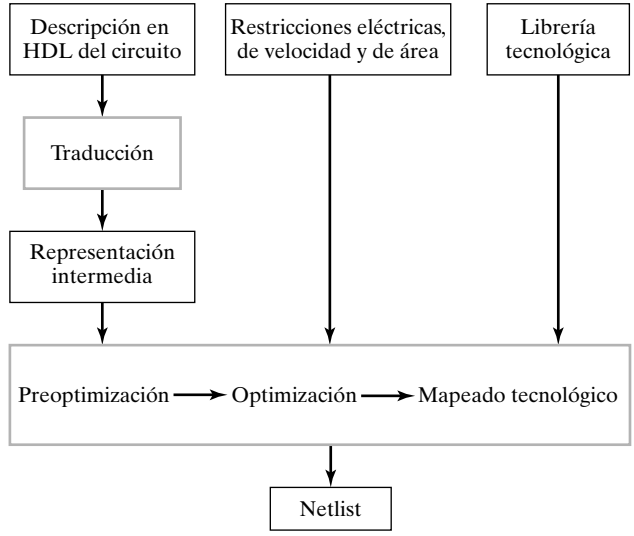
Con independencia del HDL concreto, existe un procedimiento típico a la hora de emplear una descripción HDL como entrada para la simulación. Los pasos de este procedimiento son: análisis, elaboración e inicialización, seguidos finalmente por la simulación. Normalmente, el análisis y la elaboración se efectúan por un compilador similar a los de los lenguajes de programación. Durante el *análisis*, se comprueba la descripción para detectar violaciones de las reglas sintácticas y semánticas del HDL y se genera una representación intermedia del diseño. Durante la *elaboración*, se recorre la jerarquía del diseño representada mediante esta descripción y se aplana hasta conseguir una descripción de una interconexión de módulos cada uno de los cuales esta definido únicamente por su comportamiento. El resultado final del análisis y la elaboración realizados por el compilador es un modelo de simulación de la descripción HDL original. Este modelo es entregado entonces al simulador para su ejecución. La *inicialización* fija todas las variables del modelo de simulación a los valores especificados o a sus valores por defecto. La *simulación* ejecuta el modelo de simulación en modo *batch* (por lotes) o modo interactivo empleando un conjunto de entradas que fija el usuario.

Dada la capacidad de los HDLs para describir eficiente *hardware* complejo, se puede emplear una estructura HDL especial denominada *testbench* (banco de test). El testbench es una descripción que incluye el diseño que se va a comprobar, usualmente denominado dispositivo bajo test (DUT). El testbench describe una colección de *hardware* y funciones software que aplican entradas al DUT y analizan sus salidas comprobando que sean correctas. Este enfoque, elimina la necesidad de generar por un lado las entradas del simulador y por otro analizar, usualmente a mano, la salida del simulador. La construcción de un testbench proporciona un mecanismo de verificación uniforme que puede emplearse en múltiples niveles del ciclo de diseño *top-down* para verificar el correcto funcionamiento del circuito.

Síntesis lógica

Cómo se indicó anteriormente, la disponibilidad de herramientas de síntesis lógica es la causa desencadenante del uso creciente de los HDLs. La síntesis lógica transforma la descripción RTL, expresada en HDL, de un circuito en un *netlist* optimizado formado por elementos de memoria y lógica combinacional. Posteriormente, este *netlist* se puede transformar mediante herramientas de diseño físico en el *layout* (plano o dibujo) del circuito integrado final. Este *layout* sirve como base para la fabricación del circuito integrado. Las herramientas de síntesis lógica tienen en consideración buena parte de los detalles del diseño y permiten buscar el mejor equilibrio entre coste y prestaciones, lo que resulta esencial en diseños avanzados.

En la Figura 3-4 se representa el diagrama de flujo a alto nivel de los pasos necesarios para la síntesis lógica. El usuario es quien proporciona tanto la descripción HDL del circuito a



□ **FIGURA 3-4**
Diagrama de flujo de alto nivel de una herramienta de síntesis lógica

diseñar como los distintos límites y restricciones del diseño. Las restricciones eléctricas incluyen los *fan-out* permisibles para las puertas y la carga máxima en las salidas. Las restricciones de área y velocidad dirigen las etapas de optimización de la síntesis. Normalmente, los compromisos de área fijan la máxima área permitida para el circuito dentro del integrado. También es posible indicar mediante una directiva general que debe intentar minimizarse el área. Usualmente, los requisitos de velocidad se expresan en forma de tiempos máximos de propagación permitidos a lo largo de varios caminos del circuito. También es posible, mediante el empleo de otra directiva general, indicar que debe intentar maximizarse la velocidad. Ambos, área y velocidad, tienen impacto en el coste del circuito. Normalmente, un circuito rápido tendrá una gran área y por tanto su fabricación será más costosa. Un circuito que no necesite trabajar a altas velocidades, podrá ser optimizado en área y será, en términos relativos, de fabricación más barata. En algunas herramientas de síntesis sofisticadas, el consumo de potencia también se puede especificar como una restricción. La herramienta de síntesis necesita información adicional, expresada en forma de una librería de tecnología, que describe los elementos lógicos disponibles para usar en el *netlist* así como sus parámetros de retardo y carga. Esta última información es esencial a la hora de verificar las restricciones y efectuar optimizaciones.

El primer paso en el proceso de síntesis de la Figura 3-4 es la traducción de la descripción HDL a un formato intermedio. La información en esta representación puede ser una interconexión de puertas genéricas y elementos de memoria no tomados de la librería de primitivas denominada *librería tecnológica*. Esta información también puede materializarse en un formato alternativo que representa grupos de lógica y las interconexiones entre ellos.

El segundo paso en el proceso de síntesis es la optimización. Se puede emplear un paso de preoptimización para simplificar el formato intermedio. Por ejemplo, si se detecta que en varias partes del circuito se obtiene la misma función lógica de las mismas variables, esta función sólo se implementará una única vez, y será compartida por los bloques que lo requieran. Lo siguiente es la optimización en la que el formato intermedio es procesado para intentar alcanzar las restricciones especificadas. Usualmente se realizan optimizaciones a dos niveles y a múltiples niveles. La optimización es seguida por el *mapeado tecnológico* que sustituye las puertas AND,

OR y los inversores por puertas procedentes de la librería tecnológica. Con el fin de evaluar el área y los parámetros de velocidad asociados a dichas puertas se emplea la información adicional existente en dichas librerías. Además, con herramientas de síntesis complejas, la optimización se puede aplicar durante el mapeado tecnológico aumentando de esta manera la probabilidad de alcanzar las restricciones del diseño. La optimización puede ser, para circuitos grandes, un proceso muy complejo y lento. Pueden necesitarse muchas fases de optimización para conseguir alcanzar las restricciones especificadas o para demostrar que éstas son muy difíciles, si no imposibles, de satisfacer. A fin de conseguir un diseño satisfactorio, el diseñador puede que necesite modificar estas restricciones o el HDL. Modificar el HDL puede suponer diseñar manualmente algunas partes de la lógica para alcanzar los objetivos del diseño.

Típicamente, el resultado de los procesos de mapeado tecnológico y optimización es una *netlist* que se corresponde a un esquemático compuesto por elementos de almacenamiento, puertas y otros bloques funcionales de lógica combinacional. Esta salida sirve como entrada para las herramientas de diseño físico que físicamente colocan los elementos lógicos y enrutan las interconexiones entre ellos, generando el *layout* del circuito para fabricarlo. En el caso de las piezas programables como los arrays de puertas programables en campo (FPGA) que discutiremos en la Sección 3-6, una herramienta similar a la del diseño físico genera la información binaria empleada para programar la lógica dentro de las piezas.

3-2 EL ESPACIO DE DISEÑO

Usualmente existe, para cada un diseño dado, una tecnología de implementación objetivo que determina cuáles son las primitivas disponibles y sus propiedades. Además, existe un conjunto de restricciones que deben aplicarse al diseño. En esta sección trataremos sobre las funciones de las potenciales puertas primitivas y sus propiedades, y realizaremos una breve discusión acerca de las restricciones del diseño y de los compromisos que deben considerarse a la hora de intentar alcanzarlas.

Propiedades de las puertas

Los circuitos digitales se construyen a partir de circuitos integrados. Un circuito integrado (abreviado IC) es un cristal semiconductor de silicio, coloquialmente denominado chip, que contiene componentes electrónicos como puertas lógicas o elementos de almacenamiento. Todos estos componentes están interconectados entre sí dentro del chip. Para formar un circuito integrado, el chip se monta sobre un soporte cerámico o plástico, y las conexiones se sueldan desde el chip hasta pines externos. El número de pines puede oscilar desde 14 para los encapsulados de IC más pequeños hasta varios cientos en los encapsulados mayores. Cada IC tiene un número de identificación impreso en la superficie del encapsulado que permite identificarlo. Cada fabricante publica unas hojas de datos o un catalogo con la descripción y toda la información necesaria acerca de los IC que fabrica. Normalmente, es fácil encontrar esta información en los sitios web de los diversos fabricantes.

Niveles de integración

Según la tecnología de los ICs ha ido mejorando, el número de puertas contenidas en un único chip de silicio ha aumentado considerablemente. Se acostumbra a referirse a los encapsulados

como de pequeña, mediana, gran o gran escala para diferenciar los chips formados con un número pequeño de puertas de aquellos otros formados por, desde miles hasta decenas de millones de puertas.

Integración a pequeña escala (Small Scale Integrated, SSI) se refiere a los dispositivos que contienen algunas puertas primitivas independientes en un único encapsulado. Las entradas y las salidas de las puertas se conectan directamente a los pines del encapsulado. Normalmente, el numero de puertas es menor que 10 y está limitado por el número de pines del IC.

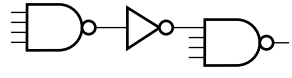
Integración a media escala (Medium Scale Integrated, MSI) se refiere a los dispositivos con aproximadamente de 10 a 100 puertas por cada encapsulado. Normalmente, son circuitos integrados capaces de realizar funciones digitales basicas como sumas de cuatro bits. Las funciones digitales MSI son parecidas a los bloques funcionales que describiremos en los Capítulos 4 y 5.

Alta escala de integración (Large Scale Integrated, LSI) son dispositivos que contienen entre 100 y algunos miles de puertas por encapsulado. LSI incluye a sistemas digitales como pueden ser pequeños procesadores, memorias pequeñas y módulos programables.

Muy alta escala de integración (Very Large Scale Integrated, VLSI) son dispositivos que pueden contener desde varios miles hasta decenas de millones de puertas por CI. Algunos ejemplos son microprocesadores complejos y los chips para procesado digital de señales. Los dispositivos VLSI han revolucionado el mundo de los sistemas digitales y el diseño de computadoras debido al pequeño tamaño de los transistores, su gran densidad y su, comparativamente, bajo coste. La tecnología VLSI ha permitido a los diseñadores crear estructuras complejas cuya fabricación, hasta ahora, no resultaba económicamente viable.

Tecnologías de circuitos

Los circuitos digitales integrados no sólo se clasifican por su función sino también lo hacen por la tecnología concreta con que se implementan. Cada tecnología dispone de sus propios dispositivos electrónicos básicos y estructuras circuitales en base a los cuales pueden desarrollase funciones y circuitos digitales mas complejos. Los distintos dispositivos electrónicos que se emplean en la construcción de los circuitos básicos son los que dan nombre a la tecnología. Así, actualmente la tecnología CMOS basada en Silicio es la que predomina debido a su alta densidad circuital, sus buenas prestaciones y su bajo consumo de potencia. Las tecnologías alternativas basadas en Arseniuro de Galio (AsGa) y Silicio-Germanio (SiGe) solo se emplean específicamente para circuitos de muy alta velocidad.



□ **FIGURA 3-5**
Implementación de una puerta NAND de 7 entradas con puertas de 4 o menos entradas.

Parámetros tecnológicos

En el diseño de un circuito electrónico existen características y parámetros diferentes según la tecnología de implementación que se emplee. Los principales parámetros que caracterizan cualquier tecnología de implementación son los siguientes:

Fan-in especifica el número de entradas disponibles en una puerta.
Fan-out especifica el número de cargas estándar atacadas por la salida de una puerta. El *fan-out* máximo para una salida determina el *fan-out* que la salida puede atacar sin afectar a

las prestaciones de dicha puerta. La carga estándar se puede definir de diversas maneras dependiendo de la tecnología empleada.

Margen de Ruido. Es el máximo nivel de voltaje de ruido externo que, superpuesto a la señal de entrada, no provoca cambios indeseados sobre la salida del circuito.

Coste. El coste de una puerta es una medida de la contribución al coste final del circuito integrado que la contiene.

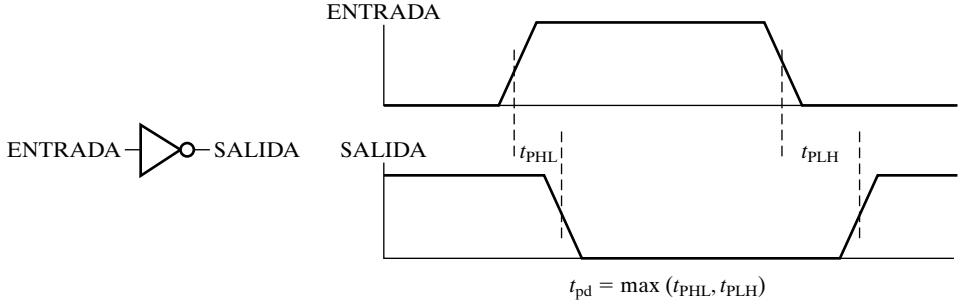
Tiempo de propagación. Es el tiempo necesario para que un cambio en una señal se propague desde la entrada a la salida de la puerta. La velocidad de funcionamiento de un circuito depende inversamente del mayor tiempo de propagación que exista a través de las puertas de dicho circuito.

Potencia disipada. Es la potencia extraída de la fuente de alimentación y consumida por la puerta. La potencia consumida se disipa en forma de calor, de modo que la capacidad de disipación se debe considerar en función de la temperatura de trabajo y de los requisitos de refrigeración del chip.

Aunque todos estos parámetros son importantes para el diseñador, a continuación se muestran más detalles de sólo algunos de ellos.

FAN-IN Para las tecnologías de alta velocidad, el *fan-in*, número de entradas de una puerta, está, a menudo, limitado en las primitivas a tan sólo cuatro o cinco. Esto se debe fundamentalmente a consideraciones electrónicas relacionadas con la velocidad de la puerta. Para lograr puertas con mayor *fan-in* se emplea, durante el proceso de mapeado tecnológico, una interconexión de puertas de menor *fan-in*. En la Figura 3-5 se muestra cómo una puerta NAND de 7 entradas se mapea empleando puertas NAND de 4 entradas y un inversor.

TIEMPO DE PROPAGACIÓN La Figura 3-6 muestra cómo se determina el tiempo de propagación. Se definen tres parámetros de propagación. El *tiempo de propagación de alto a bajo* t_{PHL}. Mide el tiempo transcurrido desde que la señal de entrada IN pasa por un nivel de tensión prefijado hasta que la salida OUT pasa por otro nivel de tensión, también prefijado, cuando la señal de salida pasa de H a L. Los niveles de tensión prefijados suelen ser del 50%, a medio camino entre el valor mínimo y máximo del nivel de voltaje de la señal; dependiendo de las familias lógicas, pueden llegar a usarse otros niveles de tensión. El tiempo de propagación de bajo a alto t_{PLH} mide el tiempo transcurrido desde que la señal de entrada IN pasa por un nivel de tensión prefijado hasta que la salida OUT pasa por otro nivel de tensión, también prefijado, en una transición de L a H de la salida. Se define el tiempo de propagación t_{pd} como el mayor de los tiempos de propagación anteriormente definidos. La razón por la que se elige el máximo de los dos

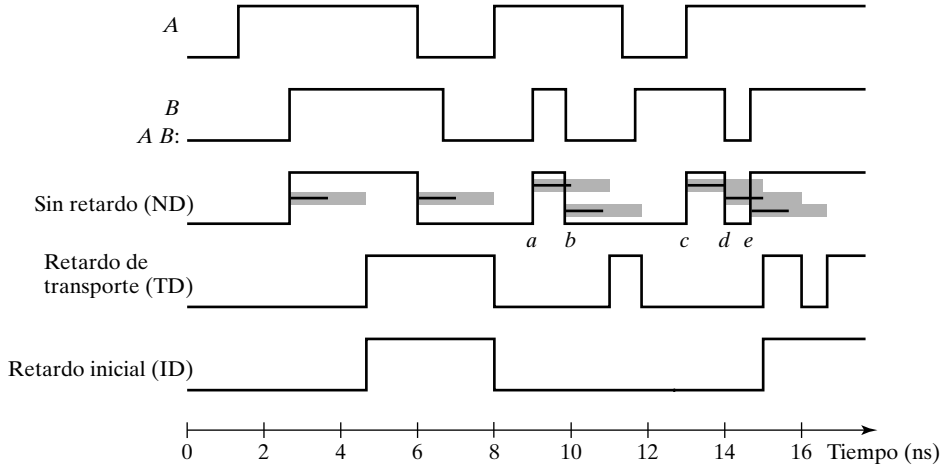


□ **FIGURA 3-6**
Tiempo de propagación para un inversor

valores es que estamos preocupados por encontrar el tiempo más largo que tarda una señal en propagarse desde las entradas a las salidas. Los fabricantes normalmente especifican, para sus productos los valores máximos y típicos tanto de t_{PHL} como de t_{PLH} y t_{pd}.

Para modelar las puertas durante la simulación se emplean dos modelos distintos: el modelo de retardo de transporte y el de retardo inercial. En el modelo de *retardo de transporte*, las variaciones en la salida como respuesta a cambios en la entrada se producen tras un determinado retardo de propagación. El modelo de *retardo inercial* es similar al de retardo de transporte, excepto que si los valores de salida cambian dos veces en un intervalo de tiempo menor que el *tiempo de rechazo*, el primero de los cambios no se reflejará. El tiempo de rechazo es un valor determinado, nunca mayor que el retardo de propagación y muchas veces igual. La Figura 3-7 muestra una puerta AND modelada mediante ambos retardos: de transporte e inercial. Con el fin de favorecer un mejor entendimiento de los comportamientos de los retardos, también se muestra la señal de salida de la puerta AND sin retardo. Una barra coloreada sobre la forma de onda de la señal, indica un tiempo de retardo de propagación de 2 ns tras cada variación en la entrada y una pequeña línea negra muestra un tiempo de rechazo de 1ns. La salida modelada con el retardo de transporte es idéntica a la que se obtuvo sin retardo, a excepción de que aparece desplazada a la derecha 2 ns. Para el caso del retardo inercial, la forma de onda también aparece desplazada. Para determinar la forma de onda de una salida con retardo, denominaremos *flanco* (*edge*) a cada cambio en una forma de onda. Con el fin de determinar si aparece un determinado flanco en la salida ID, debemos observar si existe un segundo flanco en la salida ND antes de que acabe el tiempo de rechazo para el flanco en cuestión. Dado que el flanco b ocurre antes que acabe el tiempo de rechazo para el flanco a en la salida ND, el flanco a no aparece en la salida ID. El flanco b es ignorado al no cambiar el estado de ID. Debido a que el flanco d ocurre al finalizar el tiempo de rechazo después del flanco c en la salida ND, el flanco c aparece. Por contra, el flanco e ocurre durante el tiempo de rechazo después del flanco d, con lo que el flanco d no aparece. Como el flanco c aparece y el d no, el flanco e no produce ningún cambio.

FAN-OUT Una aproximación para medir el *fan-out* es usar una carga estándar. Cada entrada de una puerta representa una carga en la salida de la puerta que la ataca y que se mide en unidades de carga estándar. Por ejemplo, la entrada de determinado inversor puede representar una carga



□ **FIGURA 3-7**
Ejemplo de comportamiento de retardos de transporte e inerciales

igual a 1.0 carga estándar. Si una puerta excita a seis de estos inversores, entonces el *fan-out* es igual a 6.0 cargas estándar. Aún más, la salida de una determinada puerta puede excitar a una carga máxima denominada máximo *fan-out*. La determinación del máximo *fan-out* es función de cada familia lógica concreta. Nuestra discusión queda limitada a CMOS, en la actualidad la más popular de las familias lógicas. Para las puertas CMOS, la carga en la salida de una puerta en un circuito integrado debida al cableado y a las entradas de otras puertas se modela como una capacidad. Esta carga capacitiva no tiene efecto sobre los niveles lógicos, como sí sucede a menudo con las cargas de otras familias lógicas. En cambio, la carga sobre la salida de una puerta determinada, sí influye en el tiempo que la salida tarda en pasar de L a H y de H a L. De modo que si la carga sobre la salida de la puerta es mayor, este tiempo, denominado *tiempo de transición*, aumenta. Así, el *fan-out* máximo de una puerta es el número de cargas estándar capacitivas que pueden excitarse sin que el tiempo de transición supere los valores máximos permitidos. Por ejemplo, una puerta con un *fan-out* máximo de 8 cargas estándar podría excitar hasta 8 inversores que presentan 1.0 carga estándar en sus entradas.

Puesto que representa capacidades, el *fan-out* real de la puerta, en términos de cargas estándar, también afecta a los retardos de propagación de la puerta. De este modo, el tiempo de propagación puede expresarse mediante una sencilla fórmula o una tabla que considere un retardo fijo más un retardo por cada carga estándar excitada, tal y como muestra el próximo ejemplo.

EJEMPLO 3-1 Cálculo del retardo de una puerta en base al *fan-out*

La salida de una puerta NAND de 4 entradas se conecta a las entradas de las siguientes puertas, cada una representando un número de cargas estándar determinado:

NOR de 4 entradas −0.80 carga estándar
NAND de 3 entradas −1.00 carga estándar y
inversor −1.00 carga estándar.

La fórmula para el retardo de la puerta NAND de 4 entradas es

$$t_{pd} = 0.07 + 0.021 \times SL \text{ ns}$$

donde *SL* es la suma de las cargas estándar excitadas por la puerta.

Despreciando el retardo por cableado, el retardo calculado para la puerta NAND cuando está cargada es

$$t_{pd} = 0.07 + 0.021 \times (0.80 + 1.00 + 1.00) = 0.129 \text{ ns}$$

En los modernos circuitos de alta velocidad, muchas veces la fracción del retardo debida a la capacidad del cableado no es despreciable. Aún siendo imprudente ignorarlo, es difícil de calcular, pues depende de la disposición del cableado en el circuito integrado. No obstante, y puesto que ahora no se dispone de esta información ni de un método capaz de proporcionar una buena estimación, aquí se ignorará esta componente del retardo. ■

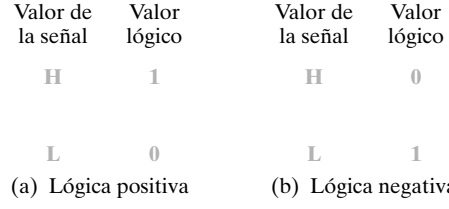
Tanto el *fan-in* como el *fan-out*, deben tenerse en cuenta durante el paso del mapeado tecnológico del ciclo de diseño Las puertas con *fan-in* mayores que los permitidos por la tecnología se deberán implementar con varias puertas. Puertas con *fan-out* mayores que el máximo permitido o que presenten retardos demasiado grandes deberán reemplazarse por varias puertas, o bien deberán añadirse buffers a sus salidas.

COSTE Normalmente, el coste de una primitiva en los circuitos integrados se calcula en base a la superficie ocupada por la célula de la primitiva en el layout del circuito. El área del layout de

la célula es proporcional al tamaño de los transistores y al de las interconexiones interiores de dicha célula. Despreciando el área debida al cableado, el área de la célula es proporcional al número de transistores que contiene, usualmente proporcional al número de entradas de dicha célula. Si se conoce el área total de un layout, entonces el valor normalizado de esta área proporciona una estimación del coste más precisa que la obtenida a partir del número de entradas.

Lógica positiva y negativa

Excluyendo las transiciones, las señales binarias de las entradas y salidas de cualquier puerta toman uno de entre dos posibles valores: H o L. Un valor representa un 1 lógico y el otro un 0 lógico. Existen dos maneras diferentes de asignar niveles de señal a valores lógicos, tal y cómo se muestra en la Figura 3-8. Si se elige el nivel alto H para representar el 1 lógico, hablamos de sistema en *lógica positiva*. Por el contrario, un sistema en *lógica negativa* escogería el nivel bajo L para representar el 1 lógico. Los términos «positiva» y «negativa» son engañosos, pues las señales pueden tener tanto tensiones positivas como negativas. No son los valores concretos de una señal los que determinan el tipo de lógica, sino más bien la *posición relativa* de los márgenes de señal asociados con cada valor lógico.



□ **FIGURA 3-8**
Señales y polaridad lógica

Las hojas de datos de los circuitos integrados definen las puertas lógicas en términos tanto de valores lógicos como de valores de señal H y L. Si se emplean H y L, será el usuario quien decida emplear lógica positiva o negativa. Consideremos, por ejemplo, la tabla de verdad de la Figura 3-9(a). Esta tabla se ha obtenido del catálogo de la puerta CMOS de la Figura 3-9(b). La tabla indica el comportamiento físico de la puerta cuando H son 5 voltios y L es 0 voltios. La tabla de verdad de la Figura 3-9(c) supone que se trabaja en lógica positiva, asignando un 1 a H y un 0 a L. La tabla se corresponde con la tabla de verdad de la operación AND. El símbolo grafico en lógica positiva de una puerta AND es el que aparece en la Figura 3-9(d).

Implementar independientemente estas siete funciones exige 27 puertas AND y 7 puertas OR. Sin embargo, compartiendo seis de los productos comunes a las diferentes expresiones de salida, el número de puertas AND se reduce a 14, y además se consigue un importante ahorro en cuanto al número total de entradas. Por ejemplo, el término $\overline{BCD}$ aparece *a*, *c*, *d*, y *e*. La salida de la puerta AND que implementa este producto va directamente a las entradas de las puertas OR de estas cuatro funciones. Para esta función detenemos la optimización en el circuito de dos niveles con puertas compartidas, comprendiendo que no será posible reducir aún más el número total de entradas si no es mediante la optimización a múltiples niveles.

Ahora vamos a trabajar en lógica negativa con la misma puerta física, asignando un 1 para L y un 0 para H. El resultado es la tabla de verdad de la Figura 3-9(e). Esta tabla representa la operación OR, aunque con las filas de la tabla desordenadas respecto a su ordenación habitual.

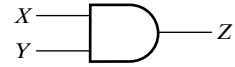
X	Y	Z
L	L	L
L	H	L
H	L	L
H	H	H

(a) Tabla de verdad con H y L

(b) Puerta de diagramas de bloques

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

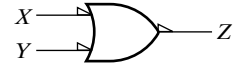
(c) Tabla de verdad en lógica positiva



(d) Puerta AND en lógica positiva

X	Y	Z
1	1	1
1	0	1
0	1	1
0	0	0

(e) Tabla de verdad en lógica negativa



(f) Puerta OR en lógica negativa

□ **FIGURA 3-9**
Demostración de lógica positiva y negativa

La Figura 3-9(f) muestra el símbolo grafico en lógica negativa de una puerta OR. Los triángulos pequeños que aparecen tanto en las entradas como en las salidas son los *indicadores de polaridad*. Siempre que aparezca un indicador de polaridad en una entrada o en una salida, significará que dicha señal se supone en lógica negativa. De este modo, la misma puerta física puede funcionar tanto como una puerta AND en lógica positiva o como una puerta OR en lógica negativa.

La conversión de lógica negativa en positiva y viceversa es una operación que intercambia 1 por 0 y 0 por 1 tanto en entradas como en salidas. Dado que el intercambio de 0 y 1 forma parte de la operación de obtención de la función dual, esta operación de conversión produce el dual de la función implementada por la puerta. De esta manera, el cambio de polaridad de todas las entradas y salidas obtiene la función dual de la dada, con todas las operaciones AND (y sus símbolos gráficos) convertidas en operaciones OR (y sus símbolos gráficos) y viceversa. Aún más, uno no debe olvidarse de incluir los indicadores de polaridad en los símbolos gráficos cuando se trabaje con lógica negativa, y debe además darse cuenta que las definiciones de polaridad de las entradas y salidas del circuito han cambiado. En este libro no emplearemos lógica negativa y asumiremos que todas las puertas trabajan en lógica positiva.

Compromisos de diseño

Con anterioridad, hemos visto que existe relación entre el *fan-out* de una puerta y los tiempos de propagación. Un *fan-out* mayor aumenta el tiempo de propagación. Por ejemplo, un circuito tiene una puerta G con un *fan-out* de 16.00 cargas estándar. El retardo a través de esta puerta, incluyendo la componente debida a las cargas estándar atacadas, es de 0.406 ns. Para reducir

este retardo, se añade un buffer a la salida de la puerta G y las 16.00 cargas estándar se conectan a la salida de dicho buffer. Ahora, la salida de la puerta G sólo ataca al buffer. El retardo para esta combinación en serie de la puerta y el buffer es de sólo 0.323 ns, consiguiendo una reducción de más del 20%. La puerta G tiene un coste de 2.00 mientras la puerta G más el buffer tiene un coste de 3.00. Estos dos circuitos muestran el compromiso entre coste y rendimiento, el más habitual de los compromisos a los que debe enfrentarse el diseñador. Mientras en este ejemplo se han empleado tan sólo dos circuitos sencillos, el compromiso coste/prestaciones puede hacerse a niveles mucho mas elevados dentro del diseño de un sistema. Estos compromisos pueden influir en la especificación del sistema y también en el enfoque a emplear para la implementación de sus funciones.

Continuando con el mismo ejemplo, el diseñador tiene dos elecciones. Si la puerta G sola fuese lo suficientemente rápida, debería ser elegida por su menor coste. Ahora bien, si no lo es, debe optarse por la puerta G más el buffer. Para poder tomar esta decisión debemos disponer de una o más restricciones para diseñar el circuito. Supongamos ahora que existe la restricción de que el tiempo de propagación máximo de entrada salida sea 0.35 ns. En tal caso, como con la puerta G sola no se cubre este requisito, deberíamos añadir el buffer. Alternativamente, suponga que la restricción es que el número máximo de unidades de área para el circuito sea 2.5. Dado que, la puerta G más el buffer no satisface esta condición, deberá elegirse para el diseño la puerta G sola. Finalmente, suponga que se espera que ambas restricciones se satisfagan en el diseño. Entonces ninguna solución de las anteriores es satisfactoria. Debemos encontrar un nuevo diseño que verifique ambas restricciones o bien, relajar las limitaciones impuestas de modo que alguno de los dos circuitos las cumpla.

A continuación se presentan algunas restricciones habituales en los circuitos:

- Tiempo de propagación máximo de entradas a salidas.
- Número máximo de unidades de área.
- Máxima potencia disipada.
- Máximo número de cargas estándar que presenta el circuito en sus entradas.
- Mínimo número de cargas estándar que el circuito debe poder atacar.

Normalmente, para un circuito determinado no se especifican todas estas restricciones. Por ejemplo, en lugar de tener, tanto restricciones de retardo como de coste, se podría fijar una restricción de retardo y minimizar el coste, respetando el resto de restricciones

3-3 CICLO DE DISEÑO

El diseño de un circuito combinacional comienza con la descripción del problema y termina en un diagrama lógico o *netlist* que describe al circuito. El proceso implica los siguientes pasos:

1. **Especificación:** se escribe la especificación del circuito en el caso de que ésta no exista.
2. **Formulación:** se localiza la tabla de verdad o las ecuaciones booleanas que definen las relaciones necesarias entre entradas y salidas.
3. **Optimización:** se aplica una optimización a dos niveles o a múltiples niveles. Se obtiene un diagrama lógico o se genera un *netlist* para el circuito resultante, a base de puertas AND, OR e inversores.
4. **Mapeado tecnológico:** se transforma el diagrama lógico o *netlist* en un nuevo diagrama o *netlist* empleando la tecnología de implementación disponible.
5. **Verificación:** se verifica el correcto funcionamiento del diseño final.

La especificación puede adoptar muchas formas, puede ser texto o una descripción en HDL y debe incluir los respectivos símbolos y nombres para las entradas y salidas. La formulación traduce la especificación a un formato que pueda ser optimizado. Normalmente, se trata de tablas de verdad o expresiones booleanas. Es importante que las especificaciones verbales se interpreten correctamente cuando se formulan tablas de verdad o expresiones. La optimización puede llevarse a cabo mediante numerosos y variados métodos, como la manipulación algebraica, el Método de Mapas de Karnaugh o los programas de simplificación basados en computadora. Para cada aplicación en particular existen criterios específicos que permiten elegir el método de optimización. Un diseño práctico ha de considerar las siguientes especificaciones: coste de las puertas a emplear, máximo tiempo de propagación permisible de una señal a través del circuito y limitaciones en el *fan-out* de cada puerta. Esto es bastante complicado ya que el coste de las puertas, sus retardos y los límites de *fan-out* no se conocen hasta la etapa de mapeado tecnológico. En consecuencia, es difícil saber cuándo el resultado final de una optimización es satisfactorio. En muchos casos, la optimización comienza por satisfacer un objetivo elemental, tal como obtener expresiones booleanas simplificadas en forma estándar para cada salida. El siguiente paso es una optimización a múltiples niveles en la que términos comunes son compartidos entre distintas salidas. Con herramientas más sofisticadas, la optimización y el mapeado tecnológico pueden interactuar para mejorar la probabilidad de satisfacer las restricciones. Es posible que sea necesario repetir los procesos de optimización y mapeado tecnológico varias veces para llegar a alcanzar dichas restricciones.

El resto del capítulo muestra el proceso del diseño a través de tres ejemplos. En lo que queda de sección realizaremos los tres primeros pasos del diseño: especificación, formulación y optimización. Luego, analizaremos las tecnologías de implementación y los dos últimos pasos en secciones aparte.

Las especificaciones de los dos primeros ejemplos son para una clase de circuitos denominados *conversores de código*, que traducen información de un código binario a otro. Las entradas del circuito son combinaciones de bits especificadas por el primer código, y por las salidas se genera la combinación de bits correspondiente al segundo código. El circuito combinacional realiza la transformación de un código a otro. El primer ejemplo es un conversor de código BCD a código exceso-3 para dígitos decimales. El otro conversor traduce código BCD a las siete señales necesarias para excitar un *display* de diodos emisores de luz (LED) de siete segmentos. El tercer ejemplo es el diseño de un comparador de igualdad de 4 bits que representa un circuito con un número elevado de entradas.

EJEMPLO 3-2 Diseño de un conversor de código BCD a exceso-3

ESPECIFIACIÓN: el código exceso-3 para un dígito decimal es la combinación binaria correspondiente al dígito decimal más 3. Por ejemplo, el código exceso-3 para el dígito decimal 5 es la combinación binaria $5 + 3 = 8$, que es 1000. El código exceso-3 tiene propiedades muy interesantes de cara a la implementación de la resta decimal.

Cada dígito BCD se representa por cuatro bits que se nombran, comenzando por el bit más significativo, como *A*, *B*, C, *D*. Cada dígito exceso-3 son cuatro bits que se nombran, del más al menos significativo como *W*, *X*, *Y*, *Z*.

FORMULACIÓN: la palabra en código exceso-3 se obtiene muy fácilmente a partir de la palabra en código BCD sin más que añadirle el binario 0011 (3). La tabla de verdad que relaciona las variables de entrada con las salidas se muestra en la Tabla 3-1. Observe cómo, a pesar que las cuatro variables de entrada BCD pueden adoptar 16 combinaciones posibles de bits, sólo se

□ **TABLA 3-1**
Tabla de verdad para el ejemplo del convertidor de código

Dígito decimal	Entradas BCD				Salidas exceso-3			
	A	*B*	*C*	*D*	*W*	*X*	*Y*	*Z*
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

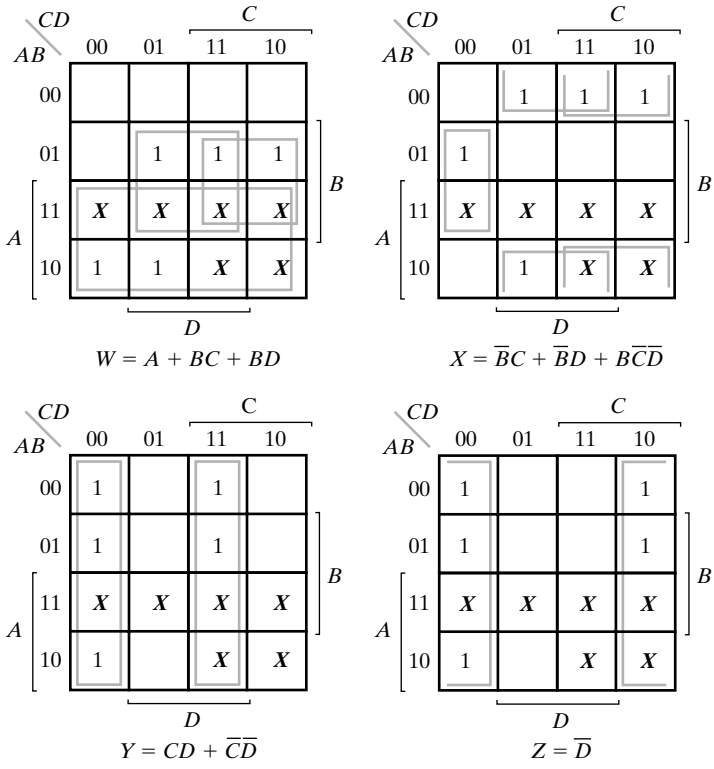
muestran 10 en la tabla de verdad. Las seis combinaciones, desde 1010 hasta 1111, no se muestran ya que no tienen significado en código BCD, y podemos asumir que nunca se producirán. Por ello, para estas combinaciones de entrada, no importa qué valores binarios asignamos a las salidas exceso-3, y por tanto, las trataremos como indiferencias.

OPTIMIZACIÓN: puesto que se trata de una función de 4 variables, usaremos los Mapas de Karnaugh de la Figura 3-10 para la optimización inicial de las cuatro funciones de salida. Los mapas se han construido para obtener expresiones booleanas de las salidas como de sumas de productos simplificadas. Cada uno de los cuatro mapas representa una de las salidas del circuito como una función de cuatro entradas. Los 1s en los mapas se obtienen directamente desde las columnas de salida de la tabla de verdad. Por ejemplo, la columna que hay debajo de la salida *W* tiene 1 para los mini términos 5, 6, 7, 8 y 9. Por tanto, el mapa para *W* debe tener 1s en los cuadros correspondientes a dichos minitérminos. Los seis minitérminos indiferentes, del 10 al 15, están marcados con una *X* en todos los mapas. Las funciones optimizadas se listan en forma de sumas de productos debajo del Mapa de Karnaugh de cada variable de salida.

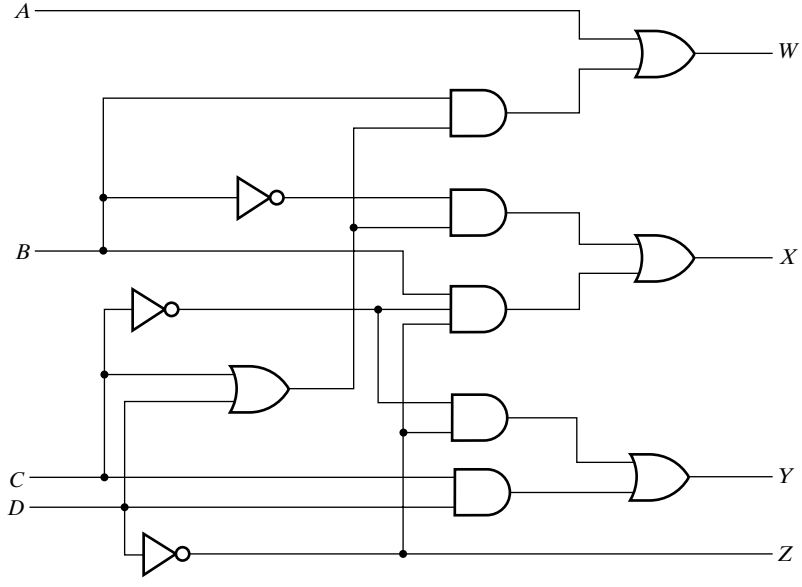
El diagrama lógico en dos niveles AND-OR puede obtenerse directamente a partir de las expresiones booleanas deducidas de los mapas. Para poder reducir el número total de entradas a las puertas, ahora 26 (incluidos inversores), realizaremos una optimización multinivel como segundo paso de la optimización La manipulación siguiente muestra una optimización de un circuito con múltiples salidas en la que se emplean tres niveles de puertas:

$$T_1 = C + D$$
$$W = A + BC + BD = A + BT_1$$
$$X = \bar{B}C + \bar{B}D + B\bar{C}\bar{D} = \bar{B}T_1 + B\bar{C}\bar{D}$$
$$Y = CD + \bar{C}\bar{D}$$
$$Z = \bar{D}$$

Esta manipulación permite que la puerta que genera $C + D$ sea compartida por la lógica de *W* y de *X*, y reduce el número total de entradas a 22. Este resultado optimizado se considera adecuado y a partir de él se obtiene el diagrama lógico de la Figura 3-11.



□ **FIGURA 3-10**
Mapas para el convertidor de BCD a exceso 3



□ **FIGURA 3-11**
Diagrama lógico del convertidor de BCD a exceso 3

■

EJEMPLO 3-3 Diseño de un decodificador BCD a siete segmentos

ESPECIFICACIÓN: los *displays* digitales encontrados en muchos productos de electrónica de consumo, como despertadores, a menudo emplean diodos emisores de luz (LEDs). Cada dígito del *display* está formado por siete segmentos LED. Cada segmento puede iluminarse mediante una señal digital. Un decodificador BCD a siete-segmentos es un circuito combinacional que acepta un dígito decimal en BCD y genera las salidas necesarias para visualizar el dígito decimal en el *display*. Las siete salidas del decodificador (*a*, *b*, *c*, *d*, *e*, *f* y *g*) seleccionan el segmento correspondiente en el *display*, tal y como muestra la Figura 3-12(a). Las representaciones numéricas elegidas para representar los dígitos decimales se muestran en la Figura 3-12(b). El decodificador BCD a siete-segmentos tiene cuatro entradas, *A*, *B*, *C* y *D* para el dígito BCD y siete salidas, de la *a* hasta la *g*, para controlar los segmentos.

(a) Designación de segmentos (b) Representación de números en el display

□ **FIGURA 3-12**
Display de 7 segmentos

FORMULACIÓN: la tabla de verdad del circuito combinacional se ilustra en la Tabla 3-2. A la vista de la Figura 3-12(b), cada dígito BCD ilumina los segmentos apropiados en el *display*. Por ejemplo, el código BCD 0011 corresponde al decimal 3, que se visualiza con los segmentos *a*, *b*, *c*, *d* y *g*. La tabla de verdad supone que un 1 lógico en una señal ilumina el segmento y que un 0 lógico lo apaga. Algunos *displays* de 7 segmentos trabajan de la manera contraria y se iluminan mediante señales a 0 lógico. En estos *displays*, las siete salidas se deben invertir. Las seis combinaciones binarias desde 1010 hasta 1111, no tienen significación en BCD. En el

□ **TABLA 3-2**
Tabla de verdad del decodificador BCD a 7 segmentos

Entradas BCD				Decodificador 7 segmentos						
A	***B***	***C***	***D***	***a***	***b***	***c***	***d***	***e***	***f***	***g***
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
Todas las demás entradas				0	0	0	0	0	0	0

ejemplo anterior, a estas combinaciones las asignamos estados indiferentes. Si hacemos lo mismo aquí el diseño producirá, probablemente, algunas visualizaciones arbitrarias y sin sentido para estas combinaciones no usadas. Siempre que estas combinaciones no se produzcan, podríamos usar este enfoque para reducir la complejidad del convertidor. Una elección más segura es apagar todos los segmentos siempre que se produzca una combinación en las entradas no usada, evitando así cualquier visualización espuria si ocurren estas entradas, pero aumentando la complejidad del convertidor. Esta elección se consigue asignando 0 a todos los minitérminos desde 10 hasta 15.

OPTIMIZACIÓN: la información de la tabla de verdad se traslada a los Mapas de Karnaugh a partir de los cuales se obtienen las primeras optimizaciones de las funciones de salida. La realización de estos mapas se deja como ejercicio. Una posible manera de simplificar las siete funciones resulta en las siguientes funciones booleanas:

$$a = \bar{A}C + \bar{A}BD + \bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}$$
$$b = \bar{A}\bar{B} + \bar{A}\bar{C}\bar{D} + \bar{A}CD + A\bar{B}\bar{C}$$
$$c = \bar{A}B + \bar{A}D + \bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}$$
$$d = \bar{A}C\bar{D} + \bar{A}\bar{B}C + \bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C} + \bar{A}B\bar{C}D$$
$$e = \bar{A}C\bar{D} + \bar{B}\bar{C}\bar{D}$$
$$f = \bar{A}B\bar{C} + \bar{A}\bar{C}\bar{D} + \bar{A}B\bar{D} + A\bar{B}\bar{C}$$
$$g = \bar{A}C\bar{D} + \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C}$$

Implementar independientemente estas siete funciones exige 27 puertas AND y 7 puertas OR. Sin embargo, compartiendo seis de los productos comunes a las diferentes expresiones de salida, el número de puertas AND se reduce a 14, y además se consigue un importante ahorro en cuanto al número total de entradas. Por ejemplo, el término $\bar{B}\bar{C}\bar{D}$ aparece *a*, *c*, *d* y *e*. La salida de la puerta AND que implementa este producto va directamente a las entradas de las puertas OR de estas cuatro funciones. Para esta función detenemos la optimización en el circuito de dos niveles con puertas compartidas, comprendiendo que no será posible reducir aún más el número total de entradas si no es mediante la optimización a múltiples niveles. ■

El decodificador BCD a siete segmentos es denominado decodificador por la mayoría de los fabricantes de circuitos integrados porque decodifica, a partir de un dígito decimal, un código binario. Sin embargo, verdaderamente, se trata de un convertidor de código ya que convierte un código decimal de 4 bits en un código de siete bits. La palabra «decodificador» está normalmente reservada para otro tipo de circuitos que se mostrarán en el próximo capítulo.

En general, el número total de puertas puede reducirse en un circuito combinacional de múltiples salidas empleando los términos comunes de las funciones de salida. Los mapas de las funciones de salida pueden ayudar a localizar los términos comunes buscando implicantes idénticos entre dos o más mapas. Algunos de estos términos comunes pueden no ser implicantes primos de las funciones individuales. El diseñador debe ser suficientemente ingenioso para combinar las células de los mapas de forma que se generen términos comunes. Esto puede realizarse de una manera más formal empleando un procedimiento de simplificación de funciones de múltiples salidas. Los implicantes primos se definen no sólo para cada función individual sino también para todas las posibles combinaciones de las funciones de salida. Estos implicantes primos se obtienen aplicando un operador AND a cada posible subconjunto no vacío de las funcio-

nes de salida y localizando los implicantes primos de cada resultado. Este procedimiento se implementa de distintas formas en los programas de simplificación lógica de las herramientas de síntesis lógica y ha sido el método empleado para obtener las ecuaciones del Ejemplo 3-3.

EJEMPLO 3-4 Diseño de un comparador de igualdad de 4-bit

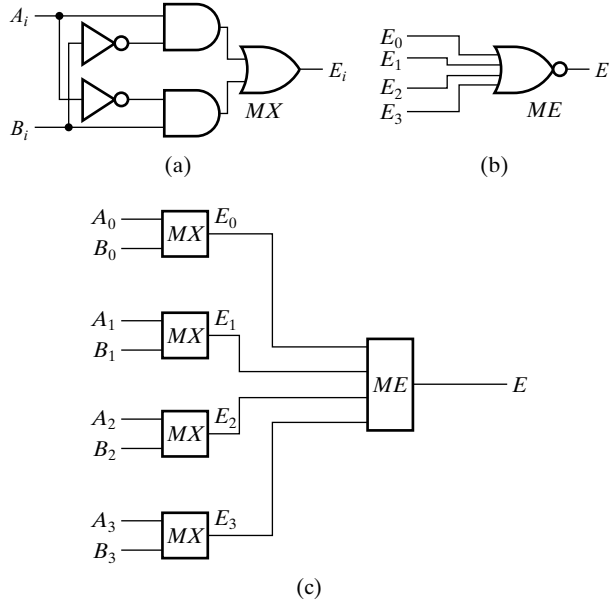
ESPECIFICACIÓN: las salidas del circuito consisten en dos vectores: A(3:0) y B(3:0). El vector A se compone de cuatro bits, A(3), A(2), A(1) y A(0), siendo A(3) el bit más significativo. El vector B tiene una definición similar sin más que reemplazar B por A. La salida del circuito es una variable E de un único bit. La salida E es igual a 1 si A y B son iguales y 0 si A y B son distintos.

FORMULACIÓN: puesto que este circuito tiene ocho entradas, resulta inviable emplear una tabla de verdad para la formulación. Para que A y B sean iguales los valores de los bits en cada una de las posiciones relativas, desde 3 hasta 0, de A y de B deben ser iguales. Si, para posiciones de bits iguales, A y B contienen los mismos valores binarios, entonces $E = 1$; de lo contrario, $E = 0$.

OPTIMIZACIÓN: en este circuito, usamos la intuición para desarrollar rápidamente un circuito multinivel usando jerarquía. Puesto que se debe comparar cada bit de A con el correspondiente bit de B, descompondremos el problema en cuatro circuitos comparadores de 1 bit y un circuito adicional que combine las salidas obtenidas de dichos circuitos comparadores para obtener E. Para el bit de posición i, definimos la salida del circuito $E_i = 0$ si $A_i = B_i$ y $E_i = 1$ si $A_i \neq B_i$. Este circuito se puede describir mediante la ecuación

$$E_i = \bar{A}_i B_i + A_i \bar{B}_i$$

cuyo diagrama esquemático se muestra en la Figura 3-13(a). Mediante el empleo de jerarquía y reutilización podemos utilizar cuatro copias de este circuito, una por cada uno de los bits de A



□ **FIGURA 3-13**
Diagrama jerárquico de un comparador de igualdad de 4 bits

y B. La salida E valdrá 1 sólo si todos los E_i valen 0. Esto puede describirse mediante la siguiente ecuación:

$$E = \overline{E_0 + E_1 + E_2 + E_3}$$

que tiene el diagrama dado en la Figura 3-13(b). Los dos circuitos dados son circuitos óptimos de dos niveles. El circuito completo se describe jerárquicamente mediante el diagrama de la Figura 3-13(c). ■

3-4 MAPEADO TECNOLÓGICO

Hay tres filosofías básicas para diseñar circuitos VLSI. En el diseño *full-custom*, se diseña el chip completamente, hasta los niveles más pequeños del layout. Puesto que este proceso es muy costoso el diseño *custom* sólo puede justificarse para ICs rápidos y de muy alta densidad que se espera vender en muy grandes cantidades.

Una técnica estrechamente relacionada con ella es el diseño con células estándar (*standard-cell*), en el que muchas partes del diseño han sido realizadas con anterioridad o posiblemente usadas en otros diseños previos. La interconexión de estas partes prediseñadas forma el diseño del IC. Esta metodología, de coste intermedio, proporciona menor densidad de integración y menores prestaciones que el diseño *full-custom*.

La tercera aproximación al diseño VLSI es el uso de arrays de puertas (*gate arrays*). En los arrays de puertas se emplea una matriz rectangular de puertas fabricadas en silicio. Esta matriz se repite cientos de veces, de modo que el chip entero contiene puertas iguales. Dependiendo de la tecnología que se emplee, es posible integrar en un único IC arrays de matrices conteniendo desde 1000 a millones de puertas. El empleo de un array de puertas necesita que el diseño especifique cómo se interconectan las puertas y como se enrutan esas interconexiones. Muchos pasos del proceso de fabricación son comunes e independientes de la función lógica final. Estos pasos, al ser usados en numerosos diseños distintos, resultan económicos. Con el fin de particularizar un array de puertas a un diseño concreto se necesitan una serie de pasos adicionales en el proceso de fabricación que sirven para determinar la interconexión concreta de las puertas. Debido a que la mayoría de los pasos de fabricación son comunes y a la capacidad para compartir los resultados de estos pasos entre varios diseños diferentes, este es el método de menor coste de entre las diversas tecnologías de implementación no programables.

Para las tecnologías de arrays de puertas y células estándar, el circuito se construye mediante la interconexión de células. La colección de células disponibles en una tecnología de implementación dada se denomina *librería de células*. Con el fin de poder diseñar en base a estas librerías de células, es necesario caracterizar cada una de dichas células, esto es, proporcionar una especificación detallada de cada célula que pueda ser empleada por el diseñador. Una librería de células convenientemente caracterizadas proporciona una base para el mapeado tecnológico de circuitos. Asociada a la librería está el procedimiento de mapeado tecnológico. En esta sección, consideraremos los procesos de mapeado tecnológico para librerías de células consistentes en (1) puertas de un único tipo, como puertas NAND, y (2) puertas de múltiples tipos. El mapeado tecnológico puede enfocarse hacia unas dimensiones determinadas del espacio de diseño, habitualmente coste y prestaciones. Por simplicidad nuestros procedimientos sólo se enfocan hacia la optimización del coste. Aún más, estos procesos son versiones rudimentarias de los algoritmos de mapeado tecnológico empleados en las herramientas de diseño asistido por computadora y sólo son apropiados para su aplicación manual en los circuitos más sencillos. Sin embargo nos proporcionarán una visión de cómo puede transformarse un diseño que emplea

puertas AND, puertas OR e inversores en diseños más efectivos en coste, empleando tipos de células soportadas por la tecnología de implementación disponible.

Especificaciones de las células

Las especificación de las células empleadas en los diseños basados en arrays de puertas y células estándar está formada por varios componentes. Estos componentes son típicamente los siguientes:

1. Un esquemático o diagrama lógico de la función de la célula.
2. Una especificación del área que ocupa la célula, a menudo normalizada respecto al área de una célula pequeña, como el área del inversor mínimo.
3. La carga de entrada, expresada en unidades de carga estándar, que cada entrada de la célula presenta a la salida que la excita.
4. Retardos desde cada entrada de la célula a cada salida (si es que existe un camino desde la entrada a la salida) incluyendo el efecto de las cargas estándar conectadas en la salida.
5. Una o más plantillas de la célula que serán empleadas durante la ejecución del mapeado tecnológico.
6. Uno o más modelos HDL de la célula.

Si las herramientas empleadas son capaces de realizar el layout automáticamente, entonces la especificación de las células también incluirá:

7. El layout completo de la célula.
8. Un layout simplificado que muestra la situación de las entradas y salidas, así como las conexiones de alimentación y masa para la célula. Este layout se empleará durante el proceso de interconexión.

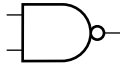
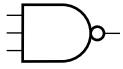
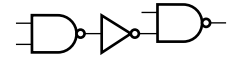
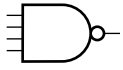
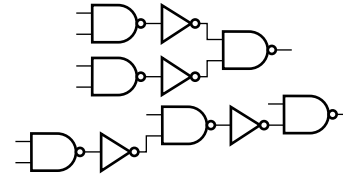
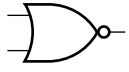
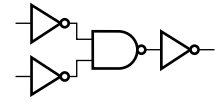
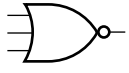
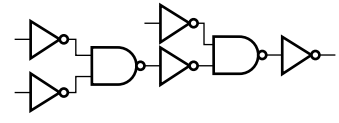
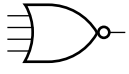
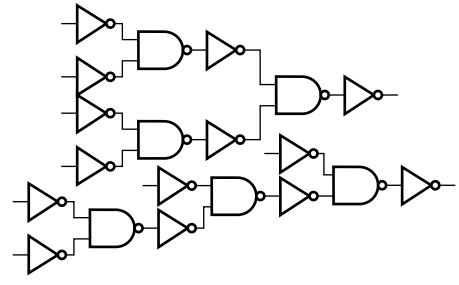
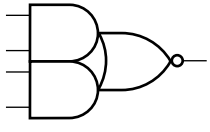
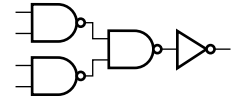
Los primeros cinco componentes listados se han incluido en una sencilla librería tecnológica de células en la próxima sub-sección. Algunos de estos componentes se discuten con más detalle.

Librerías

Para una tecnología de diseño en particular, las células se organizan en una o más librerías. Una librería es una colección de especificaciones de células. Un circuito que inicialmente consiste en puertas AND, OR y NOT se convierte, mediante el mapeado tecnológico, en otro que solamente emplea células procedentes de estas librerías. En la Tabla 3-3 se describe una librería tecnológica muy sencilla. Esta librería contiene puertas lógicas con salida negada con *fan-in* hasta 4.0 y un único circuito AOI.

La primera columna de la tabla contiene un nombre descriptivo para la célula y la segunda columna contiene el esquemático de la célula. La tercera columna contiene el área de la célula, normalizada respecto al área del inversor mínimo. Una manera sencilla de medir el coste de la célula es emplear su área. La siguiente columna proporciona la carga típica que las entradas de la célula presentan a la puerta que las excita. El valor de la carga está normalizado con respecto a una cantidad denominada carga estándar la cual, en este caso, es la capacidad presentada a un circuito por la entrada de un inversor. En el caso de las células de esta tabla, la carga de entrada es prácticamente igual para todas. La quinta columna da una sencilla ecuación lineal para calcular el retardo típico desde las entradas a las salidas para cada célula. La variable SL es la suma

□ **TABLA 3-3**
Librería de células de ejemplo para mapeado tecnológico

Nombre de la célula	Esquema de la célula	Área normalizada	Carga de entrada típica	Retardo típico	Plantillas funcionales básicas
Inverter		1.00	1.00	0.04 + 0.012 × SL	
2NAND		1.25	1.00	0.05 + 0.014 × SL	
3NAND		1.50	1.00	0.06 + 0.017 × SL	
4NAND		2.00	0.95	0.07 + 0.021 × SL	
2NOR		1.25	1.00	0.06 + 0.018 × SL	
3NOR		2.00	0.95	0.15 + 0.012 × SL	
4NOR		3.25	0.80	0.17 + 0.012 × SL	
2-2 AOI		2.25	0.95	0.07 + 0.019 × SL	

de todas las cargas estándar presentadas por las entradas de las células conectadas a la salida de la célula en cuestión. *SL* también puede incluir una estimación, expresada en cargas estándar, de la capacidad del cableado empleado en la interconexión de la salida de la célula con las entradas de las otras células. Esta ecuación muestra cómo el retardo de una célula consiste en un valor fijo al que hay que añadir el retardo debido a la capacidad de carga de la célula y que se representa por *SL*. El Ejemplo 3-5 muestra el cálculo del retardo de una célula.

EJEMPLO 3-5 Cálculo del retardo de célula

Este ejemplo muestra el efecto de la carga sobre el retardo de una célula. La salida de una NAND de 2 entradas se conecta a las siguientes células: un inversor, una NAND de 4 entradas, y una NOR de 4 entradas. En este caso la suma de las cargas estándar es

$$SL = 1.00 + 0.95 + 0.80 = 2.75$$

A partir de este valor, el retardo de la puerta NAND de dos entradas conectada a las células especificadas es

$$t_p = 0.05 + 0.014 \times 2.75 = 0.089 \text{ ns}$$ ■

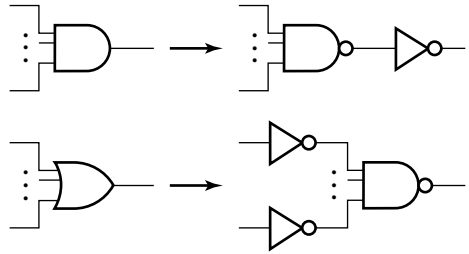
La última columna de la tabla muestra plantillas de las funciones de las células que sólo emplean como componentes funciones básicas. En este caso, las funciones básicas son: una puerta NAND de 2 entradas y un inversor. El uso de estas plantillas en funciones básicas proporciona un mecanismo para representar la función de cada célula de una forma «estándar». Tal y cómo se aprecia en las células NAND y NOR de 4 entradas, la plantilla en funciones básicas de una célula no es necesariamente única. Debe notarse, que estos diagramas representan sólo un *netlist*, y no la interconexión, la colocación o la orientación real en el *layout*. Por ejemplo, consideremos la plantilla para la puerta NAND de 3 entradas. Si la puerta NAND de la izquierda y el siguiente inversor se conectaran a la entrada de arriba de la puerta NAND de la derecha, en lugar de a la entrada de abajo, la plantilla no habría cambiado. En la próxima sección sobre técnicas de mapeado se hará más evidente el valor de estas plantillas.

Técnicas de mapeado

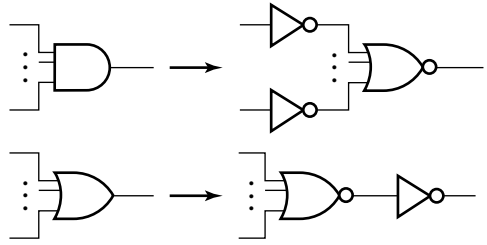
En esta sub-sección trataremos el proceso de mapeado para tecnologías no programables. Una forma razonable de implementar funciones booleanas con puertas NAND es obtener las funciones booleanas optimizadas en términos de los operadores booleanos AND, OR y NOT, y entonces mapear la función a lógica de puertas NAND. La conversión de una expresión algebraica desde AND, OR y NOT hasta NAND puede hacerse mediante un sencillo procedimiento que cambia las puertas AND y OR de los diagramas lógicos por puertas NAND. Para puertas NOR existe un procedimiento homólogo.

A partir de un circuito optimizado formado por puertas AND, OR y NOT, el siguiente procedimiento genera un circuito que emplea puertas NAND (o NOR) sin restricciones en el *fan-in*.

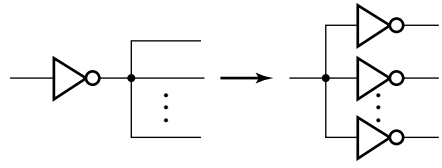
1. Reemplazar cada puerta AND y OR por su circuito equivalente en puertas NAND (o NOR) e inversores mostrado en la Figura 3-14(a) y (b).
2. Eliminar todos los pares de inversores.



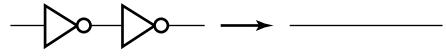
(a) Mapeado a puertas NAND



(b) Mapeado a puertas NOR



(c) Empuje de inversores a través de nodos



(d) Cancelación de pares de inversores

□ **FIGURA 3-14**
Mapeado de puertas AND, puertas OR e inversores a puertas NAND, puertas NOR e inversores

3. Sin cambiar la función lógica, (a) «empujar» todos los inversores que estén entre (i) una entrada del circuito o la salida de una puerta NAND (NOR) y (ii) las entradas de las puertas NAND (NOR) a las que se conectan, hacia estas últimas puertas. Durante este paso, cancelar cuántos pares de inversores sea posible. (b) Reemplazar los inversores en paralelo por un solo inversor que ataque todas las salidas de dichos inversores en paralelo. (c) Repetir los pasos (a) y (b) hasta que haya como máximo un inversor entre las entradas del circuito o la salida de una puerta NAND (NOR) y las entradas de las siguientes puertas NAND (NOR).

En la Figura 3-14(c), se enseña la regla para «empujar» un inversor a través de un nudo. El inversor situado en la línea que entra al nudo se sustituye por un inversor en cada una de las líneas que salen de dicho nudo. La Figura 3-14 (de) muestra el proceso de cancelación de pares de inversores basado en la identidad booleana siguiente:

$$\bar{\bar{X}} = X$$

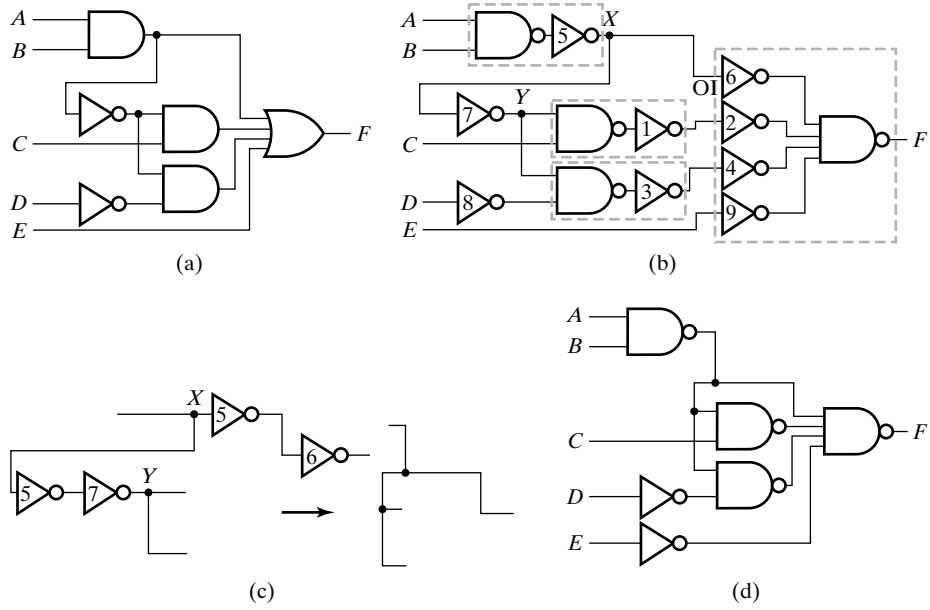
El siguiente ejemplo muestra este procedimiento para puertas NAND.

EJEMPLO 3-6 Implementación con puertas NAND

Implementar la siguiente función optimizada empleando puertas NAND:

$$F = AB + (\overline{AB})C + (\overline{AB})\overline{D} + E$$

En la Figura 3-15(a), se muestra la implementación con puertas AND, OR e inversores. En la Figura 3-15(b) hemos aplicado el paso 1 del procedimiento, reemplazando cada puerta AND y OR del circuito de la Figura 3-14(a) por sus circuitos equivalentes que emplean puertas NAND e inversores. Se han colocado etiquetas en los nudos y en los inversores para ayudar en la explicación. En el paso número 2, los pares de inversores (1, 2) y (3, 4), se han suprimido, permitiendo conectar directamente las puertas NAND, tal y como se muestra en la Figura 3-15(d). Al «empujar» el inversor 5 a través de *X*, tal y como se aprecia en la Figura 3-15 (c), hemos podido suprimir los inversores 6 y 7 respectivamente. Esto conlleva el poder conectar directamente las correspondientes puertas NAND, como ilustra la Figura 3-15(d). Al no poder aplicarse pasos similares sobre los inversores 8 y 9 el mapeado final del circuito quedará como el representado en la Figura 3-15(d). El próximo ejemplo muestra este mismo proceso para puertas NOR.



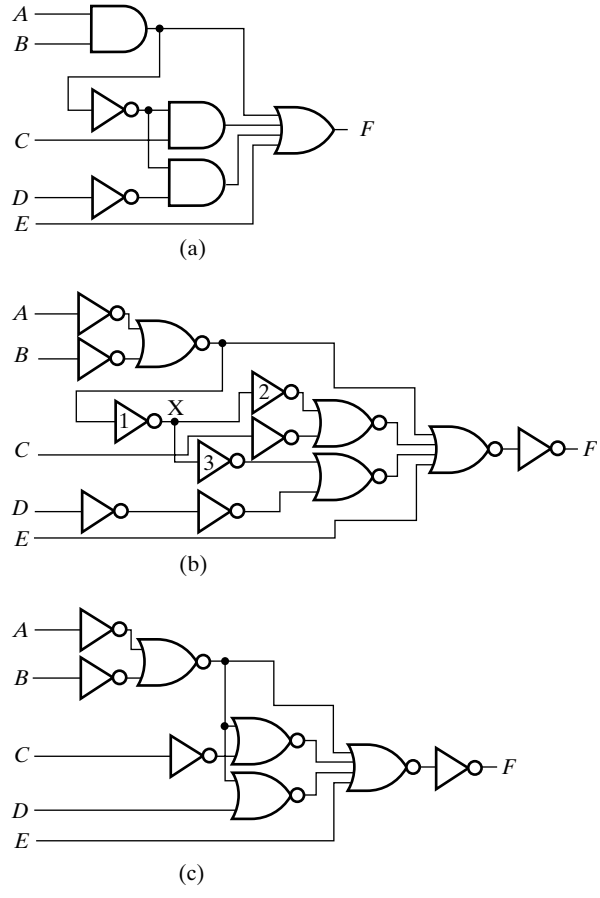
□ **FIGURA 3-15**
Solución del Ejemplo 3-6 ■

EJEMPLO 3-7 Implementación con puertas NOR

Implementar la misma función booleana optimizada usada en el Ejemplo 3-7 empleando puertas NOR:

$$F = AB + (\overline{AB})C + (\overline{AB})\overline{D} + E$$

La implementación con puertas AND, OR e inversores se muestra en la Figura 3-16(a). En la Figura 3-16(b), se aplica el paso 1 del proceso, reemplazando cada una de las puertas AND y OR del circuito de la Figura 3-14(b) por sus circuitos equivalentes que usan puertas NOR e



□ **FIGURA 3-16**
Solución del Ejemplo 3-7

inversores. Se han etiquetado los nudos y los inversores para facilitar la explicación. En el paso 2, el inversor 1 se ha «empujado» a través del nudo *X* desdoblándolo, y suprimiendo así los inversores 2 y 3 respectivamente. Del mismo modo, el par de inversores sobre la línea de entrada *D* también desaparece. Sólo los inversores sobre las líneas de entrada *A*, *B* y *C*, y el situado sobre la línea de salida *F* permanecerán, dando como resultado final el circuito mapeado que aparece en la Figura 3-16(c). ■

El coste, en número total de entradas, para el circuito mapeado del Ejemplo 3-6 es 12; mientras que en el Ejemplo 3-7, el coste es 14, por lo tanto la implementación con puertas NAND resulta menos costosa. Además, la implementación con puertas NAND tiene como máximo 3 puertas en serie, frente a las 5 puertas en serie que presenta la implementación con puertas NOR. Debido a que el circuito con puertas NOR tiene un mayor número de puertas en serie es probable que el retardo máximo, desde que un cambio en la entrada se traduce en su correspondiente cambio en la salida, sea mayor.

En el proceso anterior y en los ejemplos el objetivo del mapeado ha consistido en utilizar un único tipo de puertas, bien sean puertas NAND o puertas NOR. El siguiente proceso maneja múltiples tipos de puertas:

1. Reemplazar cada puerta AND y OR por su circuito equivalente óptimo formado sólo por puertas NAND de 2 entradas e inversores.
2. En cada línea del circuito conectada a una entrada del circuito, a una entrada de una puerta NAND, a una salida de una puertas NAND o a una salida del circuito, y siempre que esta línea no tenga inversores, insertar un par de inversores en serie.
3. Reemplazar las conexiones de puertas NAND e inversores por células disponibles en la librería, de modo que se mejore el coste en número total de entradas de los subcircuitos libres de *fan-out*. Un *subcircuito libre de fan-out* es un circuito en el que cada salida de una puerta está conectada a una única entrada. (Este paso no se cubre aquí en detalle debido a su complejidad, pero en el sitio web del libro se dispone de un ejemplo. Se han empleado las plantillas mostradas en la columna derecha de la Tabla 3-3 para encajar conexiones de puertas NAND e inversores en células disponibles en la librería.)
4. Sin cambiar la función lógica, (a) «empujar» todos los inversores que estén entre (i) una entrada del circuito o la salida de una puerta y (ii) las entradas de las puertas a las que se conectan, hacia estas últimas puertas. Durante este paso, cancelar cuántos pares de inversores sea posible. (b) Reemplazar los inversores en paralelo por un solo inversor que ataque todas las salidas de dichos inversores en paralelo. (c) Repetir los pasos (a) y (b) hasta que haya como máximo un único inversor entre las entradas del circuito o la salida de una puerta y las entradas de las siguientes puertas.

Este proceso es el fundamento del mapeado tecnológico en las herramientas de síntesis comerciales. Esta sustitución de las puertas del circuito inicial por puertas NAND de 2 entradas e inversores divide el circuito en piezas más pequeñas, proporcionando una mayor flexibilidad en el mapeado de células, lo que permitirá conseguir un resultado óptimo. El Ejemplo 3-8 muestra una forma de implementación empleando una pequeña librería de células.

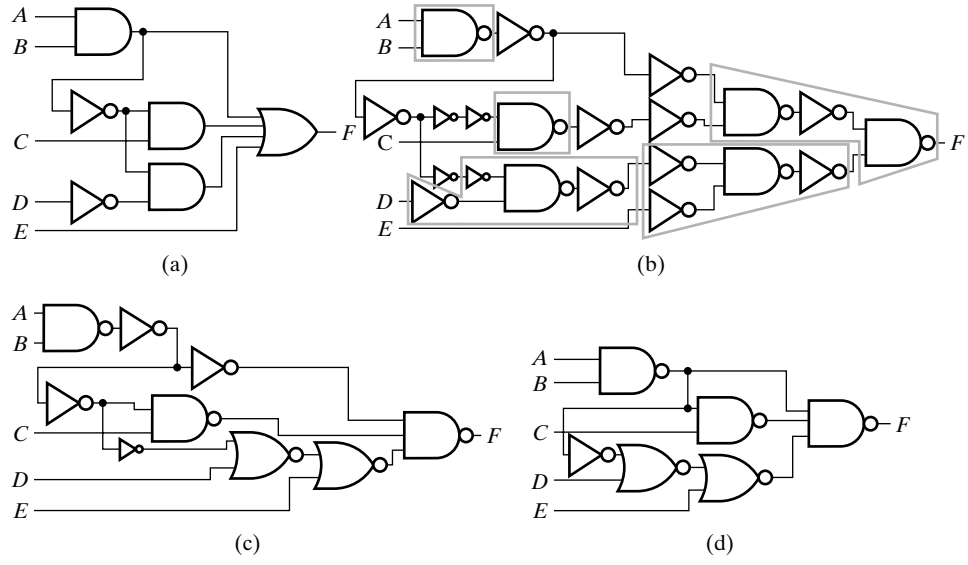
EJEMPLO 3-8 Implementación con una librería pequeña de células

Implementar la misma función booleana optimizada de los Ejemplos 3-6 y 3-7.

$$F = AB + (\overline{AB})C + (\overline{AB})\overline{D} + E$$

con ayuda de una librería de células que contiene puertas NAND de 2 entradas, puertas NAND de 3 entradas, puertas NOR de 2 entradas y un inversor. La implementación con puertas AND, OR e inversores se muestra en la Figura 3-17(a). En la Figura 3-17(b), se han aplicado los pasos 1 y 2 del procedimiento. Cada puerta AND y OR se ha sustituido por un circuito equivalente formado por puertas NAND de 2 entradas e inversores. En las líneas internas del circuito sin inversores se han insertado pares de ellos. Debido a la falta de espacio, no se muestran los pares de inversores en las entradas ni en las salidas. La Figura 3-17(c) muestra el resultado de aplicar el paso 3, el mapeado a células procedentes de la librería de células. Cada grupo de puertas NAND e inversores interconectados que aparecen enmarcados por líneas azules se ha reemplazado, empleando las plantillas de la Tabla 3-3, por una de las células disponibles en la librería. En este caso, todas las células disponibles se han usado por lo menos una vez. La aplicación del paso 4 cancela tres de los inversores, dando como resultado final el circuito mapeado que se muestra en la Figura 3-17(d). ■

La solución para el Ejemplo 3-8 tiene un coste, en número total de entradas, de 12, frente a los costes de 12 y 14 entradas de los Ejemplos 3-6 y 3-7, respectivamente. Aunque los costes en



□ **FIGURA 3-17**
Solución del Ejemplo 3-8

los Ejemplos 3-6 y 3-7 son iguales, debe notarse que las librerías de células son diferentes. En concreto, el Ejemplo 3-6 se beneficia de emplear una puerta NAND de 4 entradas que no está disponible en el Ejemplo 3-8. Sin esta célula, la solución tendría un coste adicional de dos entradas más. Así, el empleo de una librería de células más variada ha proporcionado beneficios en el coste.

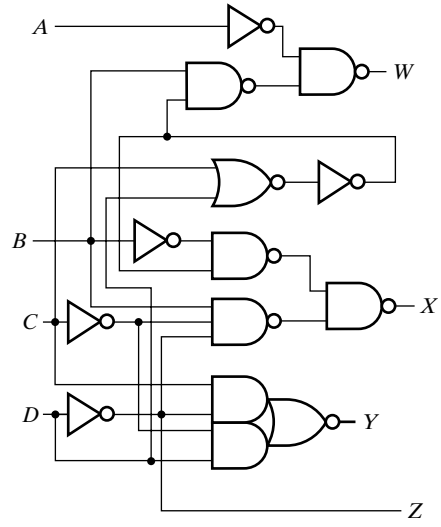
Para enlazar con los ejemplos de las primeras secciones de este capítulo, el siguiente ejemplo muestra el mapeado de un conversor de código de BCD a exceso-3 empleando una librería de células ampliada.

EJEMPLO 3-9 Mapeado tecnológico para un conversor de código de BCD a Exceso-3

El resultado final del mapeado tecnológico para un conversor de código de BCD a Exceso-3 es el dado en la Figura 3-18. El diagrama lógico inicial formado por puertas AND, OR e inversores aparece en la Figura 3-11, y la librería de células que se ha empleado es la dada en la Tabla 3-3. Como resultado de la optimización se han empleado las siguientes células de la librería: inversores, puertas NAND de 2 entradas, puertas NOR de 2 entradas y un circuito 2-2 AOI. ■

El coste en número de total de entradas para el circuito mapeado en el Ejemplo 3-9 es 22, el mismo que para el circuito original formado a partir de puertas AND, OR y NOT. El proceso de optimización, aparte de minimizar localmente los inversores, trabaja separadamente en distintas partes del circuito. Estas partes están separadas en base a los fan-out de las puertas del circuito original. La selección de estos puntos durante la optimización afectará a la calidad del resultado final. Para este circuito concreto, un circuito de partida diferente podría haber proporcionado una mejor optimización.

En general, el problema de que optimización y mapeado sean procesos separados, se maneja, en las herramientas comerciales de optimización lógica, mediante pasos combinados de optimización y mapeado.



□ **FIGURA 3-18**
Ejemplo de mapeado tecnológico: convertidor de BCD a exceso 3

EL MAPEADO TECNOLÓGICO AVANZADO Este suplemento sobre el mapeado tecnológico, incluyendo ejemplos detallados que ilustran un procedimiento de mapeado para librerías generales de células, está disponible en el Sitio Web del libro.

3–5 VERIFICACIÓN

En esta sección, se van a considerar el análisis lógico manual y el análisis lógico basado en la simulación por computadora, ambos tienen por objeto verificar el funcionamiento del circuito (es decir, determinar si un circuito dado lleva a cabo su función especificada o no). Si el circuito no responde a su especificación, entonces es incorrecto. En consecuencia, la verificación juega un papel vital evitando que un circuito diseñado incorrectamente sea fabricado y usado. El análisis lógico también puede usarse para otros propósitos como para el rediseño de un circuito y la determinación de su función.

Para verificar un circuito combinacional es esencial que la especificación sea inequívoca y correcta. Es por ello que las especificaciones dadas en forma de tablas de verdad, ecuaciones booleanas o código HDL son especialmente útiles. Inicialmente examinaremos la verificación manual continuando con los diseños de ejemplo que ya presentamos en este capítulo.

Análisis lógico manual

El análisis lógico manual consiste en hallar las ecuaciones booleanas para las salidas del circuito o, adicionalmente, encontrar la tabla de verdad para el circuito. La aproximación empleada aquí, opta por hallar las ecuaciones y entonces usarlas para encontrar la tabla de verdad. Para localizar las ecuaciones de un circuito es a menudo conveniente fragmentar el circuito en subcircuitos, definiendo variables intermedias en puntos seleccionados de él. Los puntos típicamente seleccionados serán aquéllos en los que una salida de la puerta se conecta a dos o más

entradas de otras puertas. Dichos puntos son normalmente denominados como *puntos fan-out*. Por regla general, los puntos *fan-out* de un solo inversor o de una entrada no se seleccionarán. La determinación de las ecuaciones lógicas de un circuito se ilustra usando el circuito Conversor de Código BCD-a-exceso-3 que se diseñó en las secciones anteriores.

EJEMPLO 3-10 Verificación manual de un conversor de código BCD-a-exceso-3

La Figura 3-19 muestra (a) la tabla de verdad de la especificación original, (b) la implementación final del circuito, y (c) una tabla de verdad incompleta que debe ser completada a partir de la implementación y entonces comparada con la tabla de verdad inicial. Los valores de tabla de verdad serán obtenidos a partir de las ecuaciones booleanas para *W*, *X*, *Y* y *Z* derivadas del circuito. El punto *T*1 se selecciona como una variable intermedia para simplificar el análisis:

$$T1 = \overline{\overline{C + D}} = C + D$$
$$W = \overline{\bar{A} \cdot \overline{(T1 \cdot B)}} = A + B \cdot T1$$
$$X = \overline{\overline{(\bar{B} \cdot T1)} \cdot \overline{(B \cdot \bar{C} \cdot \bar{D})}} = \bar{B} \cdot T1 + B\bar{C} \cdot \bar{D}$$
$$Y = \overline{C\bar{D} + \bar{C}D} = CD + \bar{C}\bar{D}$$
$$Z = \bar{D}$$

Sustituyendo la expresión de *T*1 en las ecuaciones de *W* y *X*, se tiene

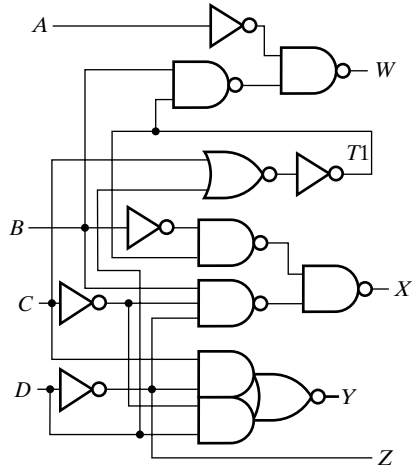
$$W = A + B(C + D) = A + BC + BD$$
$$X = \bar{B}(C + D) + B\bar{C}\bar{D} = \bar{B}C + \bar{B}D + B\bar{C}\bar{D}$$

Cada uno de los términos productos en las cuatro ecuaciones de salida puede mapearse a 1 en la tabla de verdad de la Figura 3-19(c). Se muestran los mapeados de los 1 para *A*, $\bar{B}C$, *BD*, *CD* y $\bar{D}$. Después de mapear los restantes productos a 1, las entradas en blanco se rellenan con 0. En este caso, la nueva tabla de verdad debe coincidir con la inicial, verificando que el circuito es correcto.

(a)

Entrada BCD				Salida exceso 3			
A	*B*	*C*	*D*	*W*	*X*	*Y*	*Z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

(b)



(c)

Entrada BCD				Salida exceso 3			
A	*B*	*C*	*D*	*W*	*X*	*Y*	*Z*
0	0	0	0				1
0	0	0	1				
0	0	1	0		1		1
0	0	1	1		1	1	
0	1	0	0				1
0	1	0	1	1			
0	1	1	0	1			1
0	1	1	1	1		1	
1	0	0	0	1			1
1	0	0	1	1			

□ **FIGURA 3-19**
Verificación: convertidor BCD a exceso-3 ■

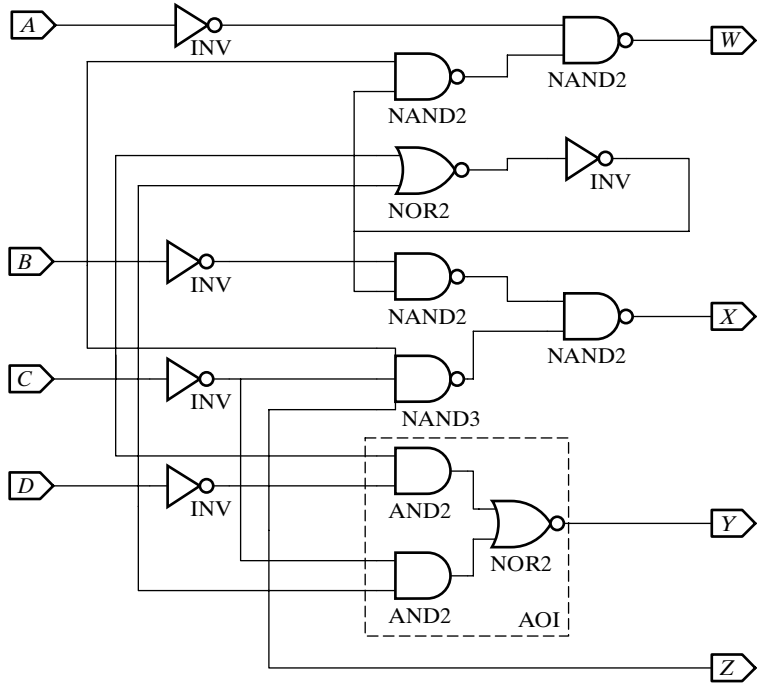
ANÁLISIS LÓGICO Este suplemento, incluyendo técnicas adicionales de análisis lógico y ejemplos, está disponible en el sitio web del libro.

La simulación

Una alternativa a la verificación manual es el uso de la simulación por computadora. El empleo de una computadora permite verificar tablas de verdad de un número significativamente más grande de variables y reduce considerablemente el tedioso esfuerzo requerido por el análisis. Dado que la simulación por computadora se basa en la aplicación de valores a las entradas, si es posible, y para conseguir alcanzar una verificación más completa, es deseable aplicar todas las posibles combinaciones de las entradas. El próximo ejemplo ilustra el uso de la herramienta Xilinx ISE4.2i para el desarrollo con FPGAs y del simulador Modelsim XE II para verificar el conversor de código BCD-a-exceso-3 verificando todas las posibles combinaciones de las entradas de la tabla de verdad.

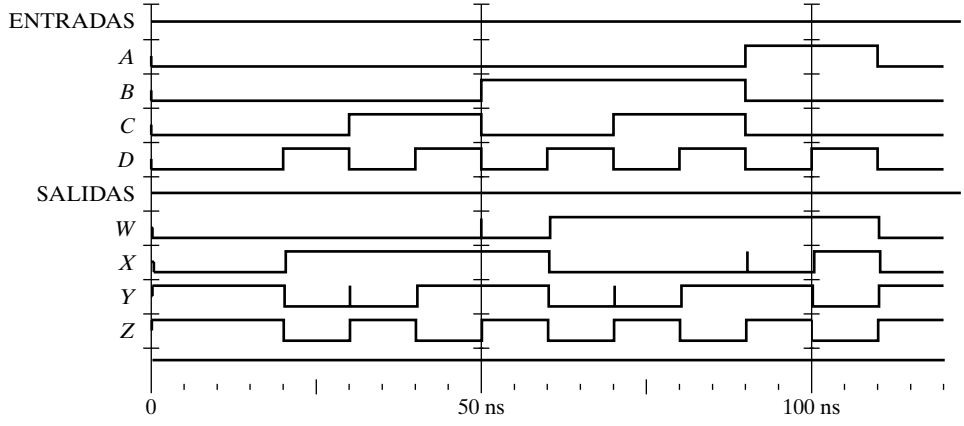
EJEMPLO 3-11 Verificación basada en simulación del conversor de código BCD-a-exceso-3

La Figura 3-19 muestra(a) la tabla de verdad de la especificación inicial, y (b) la implementación final del circuito conversor de código BCD-a-exceso-3. La implementación del circuito se ha capturado en Xilinx ISE 4.2i mediante el esquemático mostrado en la Figura 3-20. Como no hay ninguna puerta AOI en la librería de símbolos, esta puerta se ha modelado empleando las puertas disponibles. Además de introducir el esquemático también se han introducido, mediante



□ **FIGURA 3-20**
Esquemático para la simulación del convertidor BCD-a-exceso-3

formas de onda, las combinaciones de las entradas mostradas en la Figura 3-19(a). Estas formas de onda de las entradas se muestran en la sección ENTRADAS de la Figura 3-21, que ilustra los resultados de la simulación. La simulación de las formas de onda de las entradas aplicadas al circuito produce las formas de onda de salida mostradas en la sección SALIDAS. Examinando cada combinación de entrada y la combinación de salida correspondiente representada por las forma de onda, podemos verificar manualmente si las salidas coinciden con la tabla de verdad inicial. Comenzando con (A, B, C, D) = (0, 0, 0, 0) en la forma de onda de entrada, encontramos que la forma de onda de salida correspondiente es (W, X, Y, Z) = (0, 0, 1, 1). Continuando, para (A, B, C, D) = (0, 0, 0, 1), los valores para la forma de onda de salida son (W, X, Y, Z) = (0, 1, 0, 0). En ambos casos, los valores son correctos. Este proceso de verificar los valores de las formas de onda con los de las especificaciones puede continuarse para las restantes ocho combinaciones de entrada a fin de completar la verificación.



□ **FIGURA 3-21**
Ejemplo 3-10: resultados de la simulación del convertidor de BCD a exceso 3 ■

VERIFICACIÓN AVANZADA Este suplemento, conteniendo técnicas para la verificación adicionales y ejemplos, está disponible en la página web del libro.

3-6 TECNOLOGÍAS DE IMPLEMENTACIÓN PROGRAMABLES

Hasta aquí, hemos realizado una introducción a las tecnologías de implementación no programables, en el sentido de que se fabrican como circuitos integrados o interconectando circuitos integrados. En contraposición, los dispositivos lógicos programables (PLDs) se fabrican mediante estructuras que implementan funciones lógicas y estructuras que controlan su interconexionado, o que almacenan información que controla su funcionamiento, y que determinan la lógica concreta que se implementa. Estos últimos dispositivos requieren de la *programación*, un procedimiento *hardware* mediante el cual se determina qué funciones se implementarán. Las próximas tres secciones tratan de los tres tipos más simples de dispositivos lógicos programables (PLDs): la memoria de sólo lectura (ROM), el array lógico programable (PLA), y el array de lógica programable (PAL®). En un suplemento en el sitio web del libro se discuten y muestran los más complejos arrays de puertas programables en campo (FPGA). Antes de tratar los PLDs, trataremos de las tecnologías de programación en las que se apoyan. En los PLDs,

las tecnologías de programación se emplean para (1) abrir o cerrar conexiones, (2) construir las tablas de búsqueda, y (3) controlar la conmutación de transistores. Nosotros relacionaremos las tecnologías a estas tres aplicaciones.

La tecnología de programación más antigua para el control de las interconexiones es el empleo de *fusibles*. Cada uno de los puntos programables en el PLD consiste en una conexión formada por un fusible. Cuando una tensión considerablemente superior a la normal de alimentación se aplica a través del fusible, éste se quema debido a la elevada corriente, lo que abre la conexión. Los dos estados de la conexión, CERRADO y ABIERTO, se representan por un fusible intacto y uno quemado, respectivamente.

Una segunda tecnología de programación para el control de las interconexiones es la *programación con máscaras*, realizada por el fabricante del semiconductor durante las últimas fases del proceso de fabricación del chip. Las conexiones se realizan o no sobre las capas de metal que sirven como conductoras en el chip. Dependiendo de la función que se desee para el chip, la estructura de estas capas es determinada durante el proceso de fabricación. Este procedimiento es costoso ya que para cada cliente deben realizarse a medida las máscaras que generan las capas de metal. Por esta razón, la programación con máscaras sólo es rentable si se pide la fabricación de una cantidad grande de PLDs todos con la misma configuración.

Una tercera tecnología para controlar las interconexiones es el uso de *antifusibles*. Tal y cómo sugiere su nombre, el antifusible es simplemente lo contrario de un fusible. En contraste con un fusible, un antifusible consiste en un área pequeña en la que dos conductores están separados por un material de resistencia eléctrica elevada. El antifusible actúa como un camino ABIERTO antes de la programación. Al aplicar una tensión algo superior a la normal de alimentación entre los dos conductores, el material que los separa se funde o, de otra forma, su resistencia eléctrica disminuye. La baja resistencia de los materiales conductores hace que se establezca una conexión, es decir, un camino CERRADO.

Las tres tecnologías de conexión anteriores son permanentes. Los dispositivos no pueden reprogramarse, porque como resultado de la programación se han producido cambios físicos irreversibles en los dispositivos. Así, si la programación es incorrecta o necesita ser modificada, el dispositivo debe desecharse.

La última tecnología de programación que puede emplearse para el control de las interconexiones, es un elemento de almacenamiento de un solo bit que ataca la puerta de un transistor MOS de canal-N que está en el punto de programación. Si el valor del bit almacenado es un 1, entonces el transistor está conduciendo (ON), y la conexión entre fuente y drenador forma un circuito CERRADO. Para un valor del bit almacenado igual a 0, el transistor está cortado (OFF) y la conexión entre fuente y drenador es un circuito ABIERTO. Puesto que el contenido del elemento de almacenamiento puede modificarse electrónicamente, el dispositivo puede reprogramarse fácilmente. Pero para que estos valores permanezcan almacenados es necesario que no se retire la tensión de alimentación. De este modo, la tecnología de elementos de almacenamiento es volátil; es decir, la función lógica programada se pierde al retirar la tensión de alimentación.

La segunda aplicación de las tecnologías de programación es la construcción de tablas de búsqueda. Además de para controlar el interconexionado, los elementos de almacenamiento son ideales para construir estas tablas. En este caso, la combinación de entrada en la tabla de verdad se usa para seleccionar un elemento de almacenamiento, que contiene el valor de salida correspondiente en dicha tabla de verdad, y proporcionarlo como salida de la función lógica. El *hardware* consiste en (1) los elementos del almacenamiento, (2) el *hardware* para programar los valores en los elementos del almacenamiento, y (3) la lógica que selecciona el contenido de los elementos de almacenamiento que será presentado como salida de la función lógica. Puesto que

los elementos de almacenamiento se seleccionan mediante el valor de entrada, los elementos del almacenamiento combinados con el *hardware* (3) se parecen a una memoria, que almacena valores de datos que al ser seleccionados aplicando una dirección en las entradas, se presentan en las salida de dicha memoria. Así, la lógica simplemente puede llevarse a cabo guardando la tabla de verdad en la memoria —de ahí el término tabla de búsqueda (*lookup*)—.

La tercera aplicación de las tecnologías de programación es el control de la conmutación de transistores. La tecnología más popular está basada en almacenar carga en la puerta flotante de un transistor. Esta última se localiza debajo de la puerta normal de un transistor MOS y está completamente aislada por un material dieléctrico que la rodea. La carga negativa almacenada en la puerta flotante hace imposible que el transistor se ponga en conducción (ON). Si no existe carga negativa almacenada entonces es posible que el transistor conduzca si se aplica en su puerta un nivel ALTO. Ya que la carga almacenada se puede poner o quitar, estas tecnologías permiten el borrado y la reprogramación.

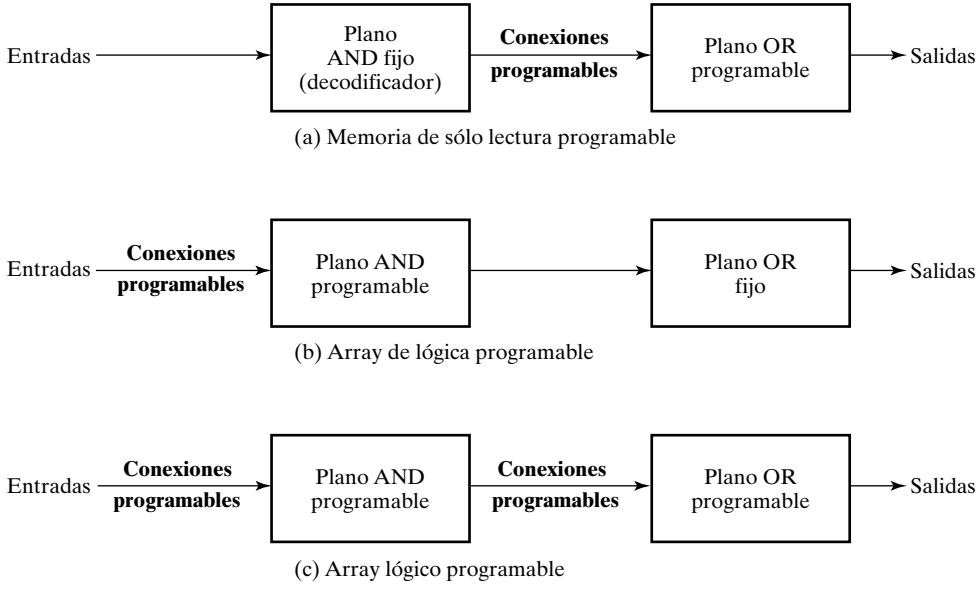
Dos de las tecnologías que emplean el control de la conmutación de transistores se denominan: *borrable* y *eléctricamente borrable*. La programación se consigue aplicando al transistor combinaciones de tensión superiores a la tensión normal de alimentación. El borrado se realiza mediante la exposición a una intensa fuente de radiación ultravioleta durante un tiempo determinado. Una vez borrados, este tipo de chips puede reprogramarse. Un dispositivo eléctricamente borrable puede borrarse mediante un proceso similar al proceso de la programación, usando tensiones superiores a los valores normales de alimentación. Puesto que controlando el transistor se evita o se permite el establecimiento de una conexión entre fuente y drenador, realmente es una forma de controlar la conexión, dando a elegir entre (1) siempre ABIERTO o (2) ABIERTO o CERRADO, dependiendo de la aplicación de un nivel ALTO o BAJO, respectivamente, en la puerta normal del transistor. Una tercera tecnología basada en el control de la conmutación del transistor es la *tecnología flash* ampliamente usada en las *memorias flash*. La tecnología flash es una forma de tecnología eléctricamente-borrable que tiene una gran variedad de opciones de borrado incluyendo el borrado de la carga almacenada en puertas flotantes individuales, de todas las puertas flotantes, o de subconjuntos específicos de puertas flotantes.

Un PLD típico puede tener de centenares a millones de puertas. Algunas, pero no todas las tecnologías lógicas programables, tienen puertas con alto *fan-in*. Para mostrar de una forma concisa el diagrama lógico interior de estas tecnologías de arrays lógicos, es necesario emplear una simbología especial para las puertas. La Figura 3-22 muestra el símbolo convencional y el símbolo de array lógico para una puerta OR de múltiples entradas. En lugar de tener múltiples líneas de entrada a la puerta, dibujamos una sola línea hacia la entrada. Las líneas de entrada se dibujan perpendiculares a esta línea y se conectan selectivamente a la puerta. Si aparece una x en la intersección de dos líneas, significa que hay una conexión. Si la x no está, entonces, no hay ninguna conexión. De manera similar, podemos dibujar la representación de array lógico de una puerta AND. Puesto que esto se hizo primero para una tecnología basada en fusibles, la representación gráfica donde quedan marcadas las conexiones seleccionadas se denomina *mapa de fusibles*. Emplearemos esta misma representación gráfica y terminología aun cuando la tecnología de programación no sea la de fusibles. Este tipo de representación gráfica para las entradas de las puertas se usará de ahora en adelante para dibujar diagramas lógicos.

(a) Símbolo convencional (b) Símbolo de array lógico

□ **FIGURA 3-22**
Símbolos convencional y de array lógico de una puerta OR

A continuación, consideraremos tres estructuras distintas de dispositivos programables. Describiremos cada una de las estructuras e indicaremos cuál es la tecnología típica, en cada caso, que se emplea para su implementación. Estos tipos de PLDs difieren en la colocación de las conexiones programables en los planos AND y OR. La Figura 3-23 muestra la situación de las conexiones para los tres tipos de dispositivos. La memoria programable de sólo-lectura (PROM) así como la memoria flash tiene un plano AND fijo construido como un decodificador y conexiones programables para las puertas OR de salida. La PROM implementa las funciones booleanas en forma de sumas de mini términos. Los arrays de lógica programable (PAL®) tienen un plano AND con conexiones programables y un plano OR fijo. Las puertas AND se programan para generar los productos de las funciones booleanas que se suman lógicamente en cada puerta OR. El más flexible de los tres tipos de PLD es el array lógico programable (PLA) que tiene conexiones programables tanto para el plano AND como para el OR. Los productos en el plano AND pueden ser compartidos por cualquier puerta OR para conseguir la implementación de los productos de sumas requeridos. Los nombres de PLA y PAL® surgieron para los distintos dispositivos de diferentes fabricantes durante el desarrollo de los PLDs.



□ **FIGURA 3-23**
Configuración básica de tres PLDs

Memorias de solo lectura

Esencialmente, una memoria de solo lectura (ROM) es un dispositivo en el que se almacena información de forma permanente. Esta información debe ser especificada por el diseñador y entonces es introducida en la ROM en forma de interconexiones o como una disposición de dispositivos. Una vez que este patrón ha sido establecido permanece dentro de la ROM incluso cuando la alimentación se apaga y se vuelve a conectar; por esto la ROM es no volátil.

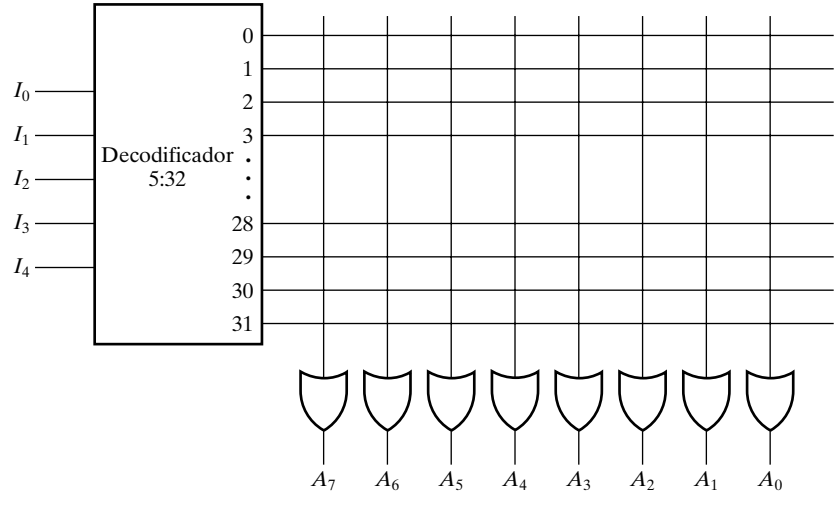
En la Figura 3-24 se muestra el diagrama de bloques de una ROM. Existen k entradas y n salidas. Las entradas seleccionan una dirección de la memoria, y por las salidas se obtienen los bits de datos de la palabra almacenada en la dirección seleccionada. El número de palabras en

□ **FIGURA 3-24**
Diagrama de bloques de una ROM

una ROM está determinado por el hecho de que con k líneas de dirección se pueden especificar 2^k palabras. Note que la ROM no dispone de líneas de entrada de datos, dado que no soporta la operación de escritura. Los chips de ROM tienen una o más entradas de habilitación y disponen también de salidas de 3 estados que facilitan la construcción de grandes arrays de ROM.

Considere, por ejemplo, una ROM de 32 × 8. Esta memoria almacena 32 palabras de ocho bits cada una. Existen cinco líneas de datos que conforman los números del 0 al 31 para cada dirección. La Figura 3-25 muestra la construcción lógica interna de esta ROM. Las cinco entradas son decodificadas a 32 salidas distintas mediante un decodificador de 5 a 32 líneas. Cada salida del decodificador representa una dirección de memoria. Las 32 salidas son conectadas a través de conexiones programables a las entradas de ocho puertas OR. El diagrama utiliza la representación de array lógico empleada en circuitos complejos (*véase* Figura 3-22). Debe considerarse que cada puerta OR tiene 32 entradas. Cada salida del decodificador se conecta a través de un fusible a una de las entradas de cada puerta OR. Puesto que cada puerta OR tiene 32 conexiones internas programables, y ya que existen ocho puertas OR, la ROM contiene $32 \times 8 = 256$ conexiones programables. En general, una ROM de $2^k \times n$ tendrá internamente un decodificador de k a 2^k líneas y n puertas OR.

Se emplearán cuatro tecnologías para la programación de la ROM. Si se emplea la programación por máscara entonces la ROM se denomina simplemente ROM. Si se emplean fusibles, la ROM puede ser programada por el usuario si dispone del equipo adecuado. En este caso, la ROM se refiere como una ROM programable o PROM. Si la ROM emplea la tecnología de puerta flotante, entonces la ROM se denomina ROM programable y borrable, o EPROM. Finalmente, si se emplea la tecnología borrable eléctricamente, la ROM se denomina ROM programable y borrable eléctricamente, o EEPROM o también E^2PROM. Como ya se dijo anterior-



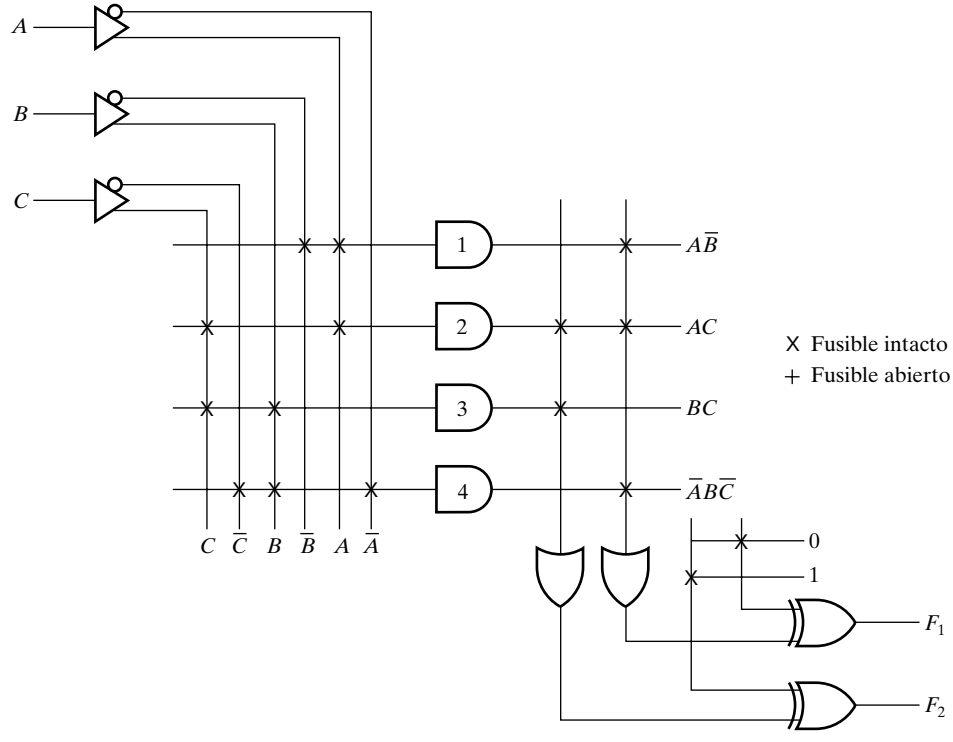
□ **FIGURA 3-25**
Lógica interna de una ROM 32 × 8

mente, la memoria flash es una versión modificada de la E^2PROM. La elección de la tecnología de programación depende de varios factores entre los que se incluye en el número de ROMs idénticas que se quieran fabricar, la volatilidad requerida, la facultad de reprogramación, y las prestaciones deseadas en términos de retardo.

Array lógico programable

El array lógico programable (PLA) es similar en concepto a la PROM, excepto que el PLA no decodifica completamente todas las variables y no genera todos los mini términos. El decodificador es sustituido por un plano de puertas AND que puede ser programado para generar productos de las variables de entrada. Estos productos son entonces conectados selectivamente a las puertas OR para generar las sumas de productos requeridas por las funciones booleanas.

En la Figura 3-26 se muestra la lógica interna de un PLA con tres entradas y 2 salidas. Este circuito es demasiado pequeño para ser eficiente en coste, pero se presenta aquí para demostrar la configuración lógica típica de un PLA. El diagrama utiliza los símbolos de array lógico empleados para circuitos complejos. Cada entrada atraviesa un buffer y un inversor, representados en el diagrama por símbolos gráficos compuestos que tienen salidas complementarias. Las conexiones programables van desde cada entrada y su complementaria a las entradas de cada puerta AND, como se indica por las intersecciones entre las líneas verticales y horizontales. Las salidas de las puertas AND tienen conexiones programables hacia las entradas de cada puerta OR. La salida de cada puerta OR entra en una puerta XOR en la que la otra entrada se puede progra-



□ **FIGURA 3-26**
PLA con tres entradas, cuatro productos y dos salidas

mar para recibir un 1 lógico o un 0 lógico. La salida es invertida cuando la entrada de la puerta se conecta a 1 (dado que $X \oplus 1 = \bar{X}$). La salida no es alterada cuando la entrada de la puerta XOR se conecta a 0 (ya que $X \oplus 0 = X$). Las funciones booleanas concretas implementadas en el PLA de la figura son

$$F_1 = A\bar{B} + AC + \bar{A}B\bar{C}$$
$$F_2 = \overline{AC + BC}$$

Los productos generados en cada puerta AND se enumeran en la salida de cada puerta en el diagrama. Cada producto está determinado por aquellas entradas que tienen su conexión cerrada. La salida de las puertas OR obtiene la suma lógica de los productos seleccionados. Esta salida puede ser complementada o no, dependiendo de la programación de la conexión asociada con la puerta XOR.

El tamaño del PLA se determina en función del número de entradas, el número de productos, y el número de salidas. Un PLA típico tiene 16 entradas, 48 productos, y 8 salidas. Para *n* entradas, *k* productos, y *m* salidas, la lógica interna de PLA consiste en *n* buffers inversores, *k* puertas AND, *m* puertas OR, y *m* puertas XOR. Existen conexiones programables entre las entradas y el plano AND, conexiones programables entre los planos AND y OR, y *n* conexiones programables asociadas a las puertas XOR.

Como ocurre con la ROM, el PLA puede ser programable por máscaras o programable en campo. En el caso de programación por máscaras, el cliente envía una tabla con la programación del PLA al fabricante. Esta tabla es utilizada por el fabricante para generar un PLA a medida que internamente tiene la lógica especificada por el cliente. Para el caso de programación en campo se emplea un PLA denominado PLA programable en campo, o FPLA. Este dispositivo puede ser programado por el usuario mediante una unidad de programación disponible comercialmente.

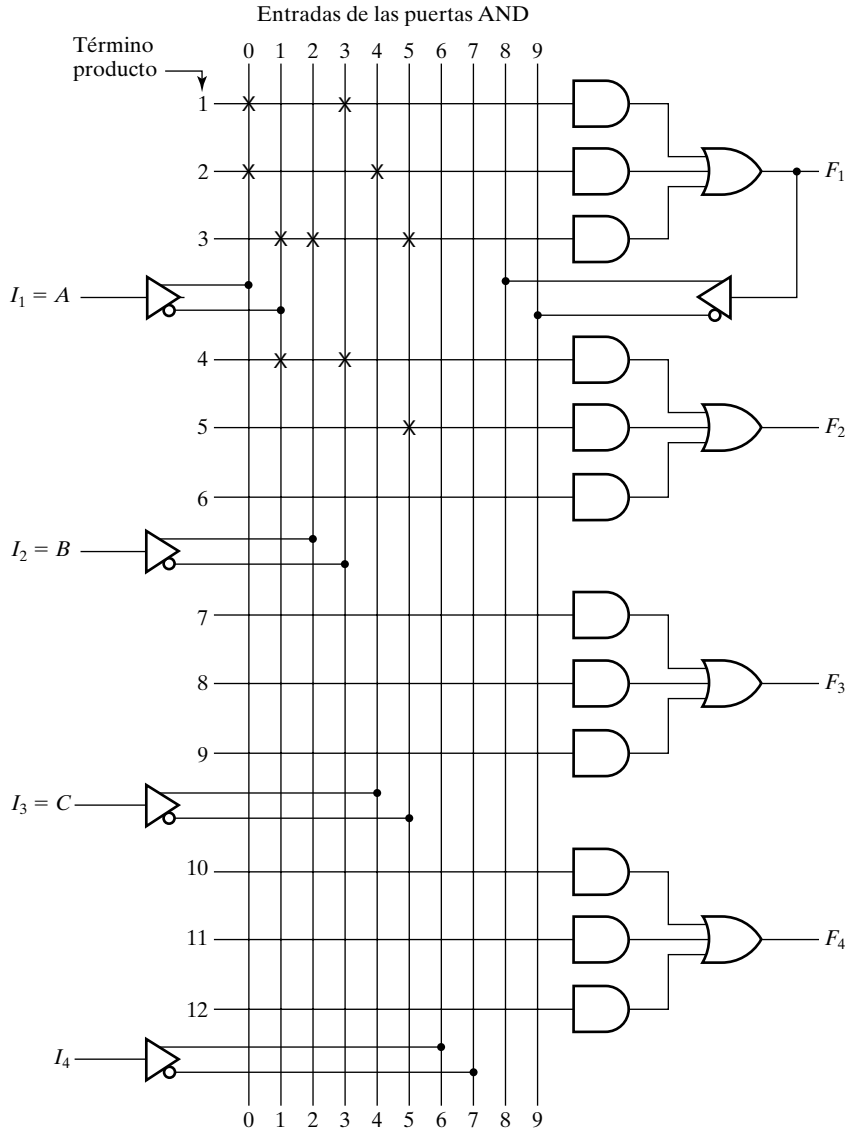
Arrays de lógica programables

El array de lógica programable (PAL®) es un PLD con un plano OR fijo y un plano AND programable. Dado que sólo es programable el plano AND, el PAL es más fácil de programar que el PLA, pero no tan flexible. La Figura 3-27 muestra la configuración lógica de un array de lógica programable típico. El dispositivo mostrado tiene cuatro entradas y cuatro salidas. Cada entrada tiene un buffer-inversor, y cada salida se genera mediante una puerta OR fija. El dispositivo tiene cuatro secciones, cada una compuesta por un array AND-OR triple, significando que hay tres puertas AND programables en cada sección. Cada puerta AND tiene 10 conexiones de entrada programables, indicadas en el diagrama mediante 10 líneas verticales que cruzan cada línea horizontal. Las líneas horizontales simbolizan las múltiples entradas de cada puerta AND. Una de las salidas mostradas se realimenta mediante un buffer-inversor hacia una de las puertas AND de entrada. Esto se hace a menudo con todas las salidas del dispositivo.

Las funciones booleanas concretas implementadas en el PAL de la Figura 3-27 son

$$F_1 = A\bar{B} + AC + \bar{A}B\bar{C}$$
$$F_2 = \overline{AC + BC} = \bar{A}\bar{B} + \bar{C}$$

Estas funciones son las mismas que las implementadas empleando el PLA. Dado que la salida complementada no está disponible, F_2 se ha expresado en forma de suma de productos.



□ **FIGURA 3-27**
Dispositivo PAL® con cuatro entradas, cuatro salidas y estructuras AND-OR triples

Los dispositivos PAL comerciales contienen más puertas que el mostrado en la Figura 3-27. Un pequeño PAL integrado puede tener hasta 8 entradas, 8 salidas y 8 secciones, cada una consistente en un array AND-OR óctuple. Cada salida de un dispositivo PAL es generada mediante un buffer de 3 estados y sirve también como entrada. Estas entradas/salidas pueden ser programadas para funcionar como entrada, como salida o como pin bidireccional, estando controlado en este caso por otra señal que actúa sobre la habilitación del buffer de 3 estados. Los PAL incluyen a menudo flip-flops entre el array y los buffers de 3 estados de las salidas. Dado que cada salida es realimentada mediante un buffer-inversor hacia las puertas AND, es sencillo implementar circuitos secuenciales.

DISPOSITIVOS LÓGICOS PROGRAMABLES VLSI Este suplemento, que cubre los fundamentos de dos típicos arrays de puertas programables en campo (FPGA) empleados a menudo en laboratorios docentes, está disponible en el la página web del libro. Este suplemento emplea multiplexores, sumadores, flip-flops, latches y SRAMs. Un apéndice del suplemento presenta una breve descripción de estos conceptos.

3–7 SUMARIO DEL CAPÍTULO

Este capítulo comenzó con la presentación de dos importantes conceptos de diseño: el diseño jerárquico y el diseño top-down, que se emplearán a lo largo del resto del libro. El diseño asistido por computadora se presentó brevemente, enfocándose a los lenguajes de descripción *hardware* y a la síntesis lógica.

En la Sección 3-2 se presentaron las propiedades de la subyacente tecnología de puertas. Se describieron dos tipos de componentes junto con un nuevo valor de salida, denominado alta impedancia (Hi-Z): los buffers de tres estados y las puertas de transmisión. En esta misma sección se han definido e ilustrado parámetros tecnológicos clave, entre los que se incluyen *fan-in*, *fan-out* y tiempo de propagación. La lógica positiva y la lógica negativa describen formas distintas de relacionar los niveles de tensión y los niveles lógicos.

El núcleo de este capítulo ha sido un ciclo de diseño de 5 pasos descrito en la Sección 3-3. Estos pasos se aplicaron tanto al diseño manual como al diseño asistido por computadora. El diseño comienza definiendo las especificaciones, siendo el siguiente paso el de formulación, en el cual la especificación se convierte en una tabla de ecuaciones. El procedimiento de optimización realiza una optimización, a dos niveles o a múltiples niveles, para obtener un circuito compuesto de puertas AND, OR e inversores. El mapeado tecnológico convierte este circuito en uno que usa eficientemente las puertas disponibles en la tecnología de implementación empleada. Finalmente se efectúa una verificación para asegurar que el circuito final satisface las especificaciones iniciales. Los tres primeros pasos de este proceso se han ilustrado mediante tres ejemplos.

Con el fin de discutir el mapeado tecnológico se presentaron las tecnologías de implementación no programables incluyéndose: *full-custom*, *standard-cell* y *gate-arrays*. También se ha presentado la especificación de células y las librerías de células, así como técnicas de mapeado tecnológico similares a las usadas por las herramientas CAD, ilustradas tanto para el caso de trabajar con un único tipo de puertas como con varios tipos de puertas.

La sección final del capítulo se centró en las tecnologías de lógica programable. Tres tecnologías básicas —memorias de sólo lectura, arrays lógicos programables y arrays de lógica programable— proporcionaron distintas alternativas para el mapeado tecnológico.

REFERENCIAS

1. HACTEL, G., and F. SOMENZI: *Logic Synthesis and Verification Algorithms*. Boston: Kluwer Academic Publishers, 1996.
2. DE MICHELI, G.: *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, Inc., 1994.
3. KNAPP, S.: *Frequently-Asked Questions (FAQ) About Programmable Logic* (http://www.optimagic.com/faq.html). OptiMagicTM, Inc., ©1997-2001.

4. LATTICE SEMICONDUCTOR CORPORATION: *Lattice GALs*(R)(http://www.latticesemi.com/products/spld/GAL/index.cfm). Lattice Semiconductor Corporation, ©1995-2002.

5. TRIMBERGER, S. M. (Ed.): *Field-Programmable Gate Array Technology*. Boston: Kluwer Academic Publishers, 1994.

6. XILINX, INC.: *Xilinx Spartan™-IIE Data Sheet* (http://direct.xilinx.com/bvdocs/publications/ds077_2.pdf). Xilinx, Inc. ©1994-2002.

7. ALTERA(R) CORPORATION: Altera FLEX 10KE Embedded *Programmable Logic Device Family Data Sheet* ver. 2.4 (http://www.altera.com/literature/ds/dsf10ke.pdf). Altera Corporation, ©1995-2002.

PROBLEMAS

El símbolo (+) indica problemas más avanzados y el asterisco (*) indica que la solución se puede encontrar en el sitio web del libro: http://www.librosite.net/Mano.

3-1. Diseñe un circuito que implemente el siguiente par de ecuaciones booleanas:

$$F = A(C\bar{E} + DE) + \bar{A}D$$
$$F = B(C\bar{E} + DE) + \bar{B}C$$

Para simplificar el dibujo del esquemático, emplee jerarquía basándose en la factorización mostrada para las funciones. Se utilizarán tres instancias (copias) de un único circuito compuesto de 2 puertas AND, una puerta OR y un inversor. Dibuje el diagrama lógico para este componente y para el circuito completo, empleando un símbolo para el componente.

3-2. Un componente que implementa la función

$$H = \bar{X}Y + XZ$$

debe emplearse, junto con inversores, para obtener la función:

$$G = \bar{A}\bar{B}C + \bar{A}BD + A\bar{B}\bar{C} + AB\bar{D}$$

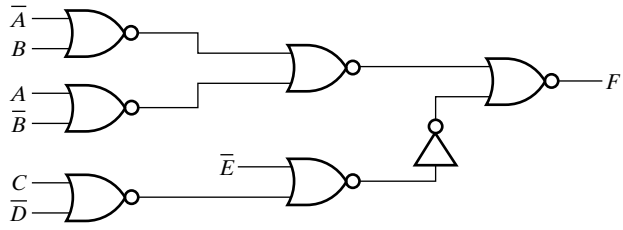
El circuito puede obtenerse mediante la aplicación del Teorema de expansión de Shannon,

$$F = \bar{X} \cdot F_0(X) + X \cdot F_1(X)$$

en la que $F_0(X)$ es $F(X)$ evaluada para $X = 0$ y $F_1(X)$ es evaluada para $X = 1$. Esta expansión para F puede implementarse para H haciendo $Y = F_0$ y $Z = F_1$. El teorema de expansión puede entonces ser aplicado a F_0 y F_1 seleccionando una variable de cada una, preferiblemente una variable que aparezca tanto negada como sin negar. Este proceso se puede iterar hasta que todas las F_i sean constantes o literales simples. Para G, emplee $X = A$ para encontrar G_0 y G_1 y entonces haga $X = B$ en G_0 y G_1. Dibuje el diagrama de G empleando un símbolo para H.

3-3. Una familia lógica de circuitos integrados tiene puertas NAND con un fan-out de 8 cargas estándar y buffers con un fan-out de 16 cargas estándar. Esboce un esquema que muestre cómo la señal de salida de una única puerta NAND puede aplicarse a 38 entradas de puertas usando tantos buffers como sea necesario. Suponga que cada entrada es una carga estándar.

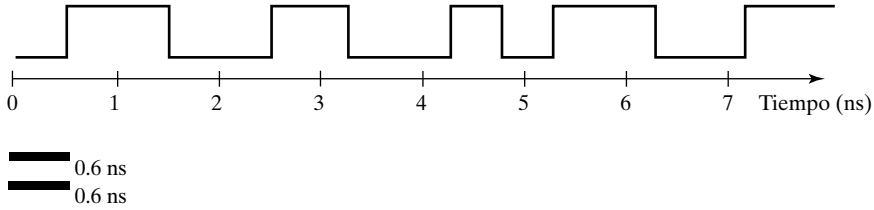
3-4. *La puerta NOR de la Figura 3-28 tiene un tiempo de propagación $t_{pd} = 0.078$ ns y el inversor tiene un retardo de propagación $t_{pd} = 0.052$ ns. ¿Cuál será el retardo de propagación del camino más largo del circuito?



□ **FIGURA 3-28**
Circuito del Problema 3-4

3-5. La forma de onda de la Figura 3-29 se aplica a un inversor. Calcule la salida del inversor, suponiendo que

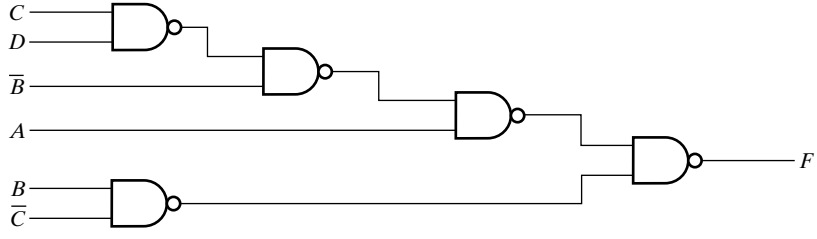
(a) no tiene retardo.
(b) tiene un retardo de transporte de 0.06 ns.
(c) tiene un retardo inercial de 0.06 ns con un tiempo de rechazo de 0.06 ns.



□ **FIGURA 3-29**
Forma de onda para el Problema 3-5

3-6. Suponiendo que t_{pd} es la media de t_{PHL} y t_{PLH}, calcule el retardo desde cada entrada hasta cada salida en la Figura 3-30

(a) Calculando t_{PHL} y t_{PLH} para cada camino, suponiendo que $t_{PHL} = 0.30$ ns y $t_{PLH} = 0.50$ ns para cada puerta. A partir de estos valores, calcule t_{pd} para cada camino.
(b) Tomando $t_{pd} = 0.40$ ns para cada puerta.
(c) Compare sus respuestas de la Sección (a) y (b) y comente las diferencias.



□ **FIGURA 3-30**
Circuito para el Problema 3-6

3-7. +El tiempo de rechazo para el retardo inercial debe ser menor que el tiempo de propagación. En los términos dados en la Figura 3-7. ¿Por qué es esta condición necesaria para determinar el valor de la salida?

3-8. +Una determinada puerta tiene $t_{PHL} = 0.05$ ns y $t_{PLH} = 0.10$ ns. Suponga que a partir de esta información se debe desarrollar un modelo de retardo inercial para el comportamiento típico del retardo de la puerta.

(a) Suponga un pulso de salida positivo (L H L), ¿qué valdrían el tiempo de propagación y el tiempo de rechazo?

(b) Comente la aplicabilidad de los parámetros encontrados en (a) suponiendo un pulso negativo de salida (H L H).

3-9. *Demuestre cómo una puerta NAND en lógica positiva es una puerta NOR en lógica negativa y viceversa.

3-10. Una función mayoría toma el valor de salida 1 si hay más 1 que 0 en sus entradas. En caso contrario, toma el valor 0. Diseñe una función mayoría de 3 entradas.

3-11. *Calcule una función que detecte un error en la representación de un dígito decimal en BCD. En otras palabras, escriba una ecuación con salida 1 cuando las entradas sean una cualquiera de la seis combinaciones no usadas en código BCD, y valor 0, en el resto de los casos.

3-12. Diseñe un conversor de código de Exceso 3 a BCD que muestre el código de salida 0000 para todas las combinaciones de entrada no válidas.

3-13. **(a)** Un sistema de iluminación a baja tensión emplea lógica de control binaria para una determinada luminaria. Esta luminaria está en una intersección en forma de T en un vestíbulo. Hay un conmutador para esta luz en cada uno de los tres puntos del final de la T. Estos interruptores tienen salidas binarias 0 y 1 dependiendo de su posición y se nombran como X_1, X_2 y X_3. La luz se controla mediante un amplificador conectado a un tiristor. Cuando Z, la entrada del amplificador, está a 1, la luz se enciende y cuando Z está a 0, la luz se apaga. Debe encontrar una función $Z = F(X_1, X_2, X_3)$ de modo que si cualquiera de los interruptores cambia, el valor de Z cambia pasando la luz de estar encendida a estar apagada.

(b) La función Z no es única. ¿Cuántas funciones Z diferentes hay?

3-14. +Un semáforo de una intersección emplea un contador binario para producir la siguiente secuencia de combinaciones en sus líneas A, B, C y D: 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000. Después de 1000, la secuencia se repite, comenzando de nuevo con 0000. Cada combinación se presenta durante 5 segundos antes que la próxima aparezca. Estas líneas conectan la lógica combinacional con las salidas de las lámparas RNS (Rojo - Norte/Sur), ANS (Amarillo - Norte/Sur), VNS (Verde - Norte/Sur), REO (Rojo - Este/Oeste), AEO (Amarillo - Este/Oeste), y VEO (Verde - Este/Oeste). La lámpara controlada por cada salida se enciende al aplicar un 1 y se apaga al aplicar un 0. Para una dirección dada, suponga que el verde está encendido durante 30 segundos, el amarillo durante 5 segundos y el rojo durante 45 segundos. (el intervalo del rojo se superpone durante 5 segundos). Divida los 80 segundos disponibles para cada ciclo entre las 16 combinaciones de los 16 intervalos y determine qué lámparas deberán lucir en cada intervalo basadas en el comportamiento esperado del conductor. Suponga que, para el intervalo 0000, acaba de ocurrir un cambio y

que VNS = 1, REO = 1 y las demás salidas están a 0. Diseñe el esquema lógico necesario para producir las seis salidas usando puertas AND, OR e inversores.

3-15. Diseñe un circuito combinacional que genere, a partir de un número de 3 bits, un número de 6 bits igual al cuadrado del número de entrada.

3-16. + Diseñe un circuito combinacional que genere, a partir de un número de 4 bits, un número de 3 bits que aproxime la raíz cuadrada del número de entrada. Por ejemplo, si la raíz es igual o mayor que 3.5 debe obtener 4. Si la raíz es menor que 3.5 y mayor o igual que 2.5 debe obtener 3.

3-17. Diseñe un circuito con una entrada BCD de 4 bits (*A, B, C, D*) que genere una salida *W, X, Y, Z* que debe ser igual a la entrada más 6 en binario. Por ejemplo, 9 (1001) + 6 (0110) = 15 (1111).

3-18. Un sistema de medida del tráfico, que se emplea para regular el acceso de vehículos desde una vía de servicio a una autopista, presenta las siguientes especificaciones para una parte de su controlador. Existen tres carriles en la vía de servicio, cada uno con su propia luz de parada (rojo) o acceso libre (verde). Uno de estos carriles, el central, tiene prioridad (en verde) sobre los otros dos. En caso contrario se aplicará un esquema «*round-robin*» a los otros dos carriles, de forma que la luz verde se alternará entre uno y otro (izquierdo y derecho). Debe diseñarse la parte del controlador que determina cual de las luces es verde (en vez de roja). Las especificaciones de este controlador son:

Entradas:

SC -Sensor de vehículo en el carril central (hay vehículo - 1, no hay - 0)
SI -Sensor de vehículo en el carril izquierdo (hay vehículo - 1, no hay - 0)
SD -Sensor de vehículo en el carril derecho (hay vehículo - 1, no hay - 0)
RR -Señal del *round robin* (izquierdo - 1, derecho - 0)

Salidas:

LC -Luz del carril central (verde - 1, roja - 0)
LI -Luz del carril izquierdo (verde - 1, roja - 0)
LD -Luz del carril derecho (verde - 1, roja - 0)

Funcionamiento:

1. Si hay un vehículo en el carril central LC es 1.
2. Si no hay vehículos en el carril central ni en el derecho entonces LI es 1.
3. Si no hay vehículos en el carril central ni en el izquierdo, pero los hay en el carril derecho es LD = 1.
4. Si no hay vehículos en el carril central, pero los hay en los dos carriles laterales, entonces si RR es 1 será LI = 1.
5. Si no hay vehículos en el carril central, pero los hay en los dos carriles laterales, entonces si RR es 0 será LD = 1.
6. Si cualquiera de LC, LI o LD no se ha especificado a 1 en alguno de los puntos anteriores, entonces es que vale 0.

(a) Localice la tabla de verdad del controlador.
(b) Localice una implementación mínima de varios niveles, que implemente esta función minimizando el número total de entradas y empleando puertas AND, OR e inversores.

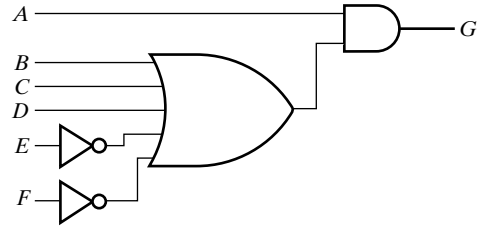
3-19. Complete el diseño del decodificador de BCD a 7 segmentos realizando los siguientes pasos:

(a) Dibuje los 7 mapas para cada una de las salidas, de acuerdo a lo especificado en la Tabla 3-2.
(b) Simplifique las 7 funciones de salida en forma de sumas de productos y determine el número total de entradas necesarias para implementar el circuito.
(c) Verifique que las 7 funcionas dadas en el texto son simplificaciones válidas. Compare su número total de entradas con el del apartado (b) y explique las diferencias.

3-20. + Se necesita una puerta NAND de 8 entradas. Para cada uno de los siguientes casos minimice el número de puertas empleadas en la solución final a múltiples niveles:

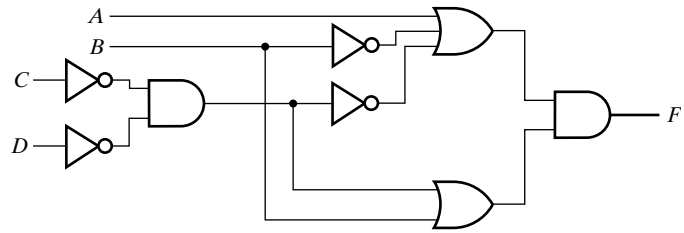
(a) Diseñe la puerta NAND de 8 entradas empleando puertas NAND de 2 entradas e inversores.
(b) Diseñe la puerta NAND de 8 entradas empleando puertas NAND de 2 entradas, puertas NOR de 2 entradas y, sólo en caso necesario, inversores.
(c) Compare el número de puertas necesario parta los Apartados (a) y (b).

3-21. Realice un mapeado tecnológico, empleando las células NAND e inversores de la Tabla 3-3, para el circuito de la Figura 3-31 minimizando el coste (mida el coste en área total normalizada).



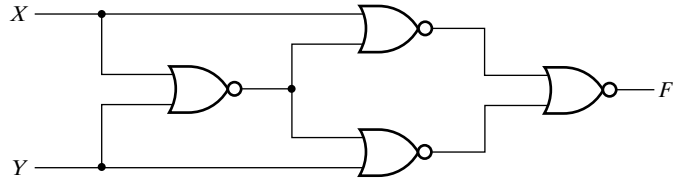
□ **FIGURA 3-31**
Circuito para el Problema 3-21

3-22. Realice un mapeado tecnológico, empleando células de la Tabla 3-3, para el circuito de la Figura 3-32 minimizando el coste (mida el coste en área total normalizada).



□ **FIGURA 3-32**
Circuito para el Problema 3-22

3-23. Empleando métodos manuales verifique que el circuito de la Figura 3-33 implementa una función XNOR.

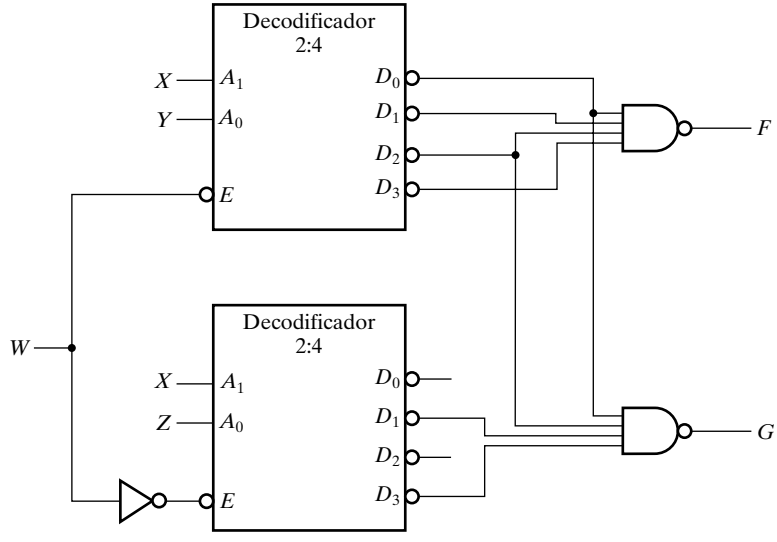


□ **FIGURA 3-33**
Circuito para el Problema 3-23

3-24. *Verifique manualmente que las funciones para las salidas *F* y *G* del circuito jerárquico de la Figura 3-34 son

$$F = \bar{X}Y + X\bar{Y}Z + XY\bar{Z}$$
$$G = XZ + \bar{X}YZ + \bar{X}\bar{Y}\bar{Z}$$

En la Figura 4-10 se encuentra el diagrama y la tabla de verdad del decodificador.



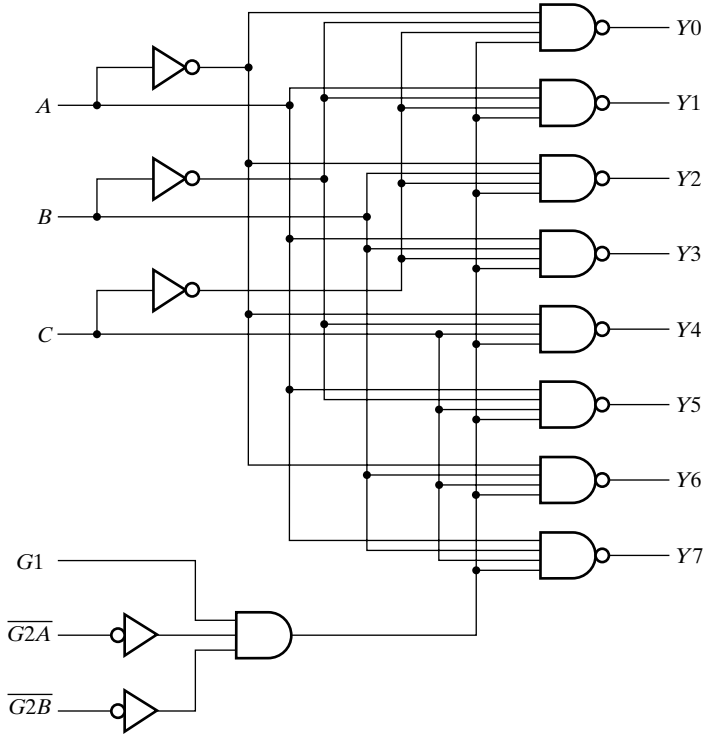
□ **FIGURA 3-34**
Circuito para los Problemas 3-24 y 3-25

3-25. Verifique manualmente que las tablas de verdad para las salidas *F* y *G* del circuito jerárquico de la Figura 3-34 son:

W	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
X	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Y	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Z	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
F	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
G	1	1	0	0	1	1	0	0	0	1	0	1	0	1	0	1

En la Figura 4-10 se encuentra el diagrama y la tabla de verdad del decodificador.

3-26. La Figura 3-35 muestra el diagrama lógico de un circuito CMOS MSI 74HC138. Localice las funciones booleanas de cada una de sus salidas. Describa con detalle el funcionamiento del circuito.



□ **FIGURA 3-35**
Circuito para los Problemas 3-26 y 3-27

3-27. Repita el Problema 3-26 empleando un simulador lógico para obtener las formas de onda de salida o una tabla de verdad parcial, en vez de obtener las expresiones lógicas del circuito.

3-28. En la Figura 3-21 se muestran los resultados de la simulación del convertidor de BCD a exceso 3 para las entradas BCD del 0 al 9. Realice una simulación similar para determinar las salidas del circuito ante las entradas BCD del 10 al 15.

CAPÍTULO

4

FUNCIONES Y CIRCUITOS COMBINACIONALES

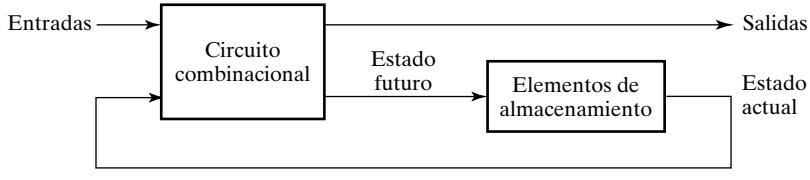
En este capítulo, estudiaremos varias funciones y sus correspondientes circuitos fundamentales, muy útiles en el diseño de circuitos digitales más grandes. Los circuitos fundamentales, reutilizables, que denominaremos bloques funcionales, implementan funciones de una única variable, decodificadores, codificadores, conversores de código, multiplexores y lógica programable. Aparte de ser bloques importantes en la construcción de circuitos y sistemas más grandes, muchas de estas funciones están fuertemente unidas a los distintos componentes de los lenguajes de descripción hardware y sirven como vehículo para la presentación del HDL. Se introducirán los lenguajes de descripción hardware VHDL y Verilog como una alternativa a las tablas de verdad, ecuaciones, y esquemáticos.

En el diagrama de una computadora genérica que aparece al principio del Capítulo 1, los multiplexores son muy importantes para seleccionar los datos en el procesador, en la memoria, y en las placas de I/O. Los decodificadores se emplean para seleccionar las placas conectadas al bus de entrada/salida y para descifrar las instrucciones a fin de determinar las funciones ejecutadas por el procesador. Los codificadores se usan en varios componentes, como el teclado. La lógica programable se usa para manejar instrucciones complejas dentro de los procesadores así como en muchos otros componentes de la computadora. En general, los bloques funcionales son muy utilizados, tanto que los conceptos de este capítulo son aplicables a la mayoría de los componentes de la computadora genérica, incluso a las memorias.

4-1 CIRCUITOS COMBINACIONALES

En el Capítulo 3, se definieron e ilustraron los circuitos combinacionales y su diseño. En esta sección definiremos algunas funciones combinacionales determinadas junto con sus correspondientes circuitos combinacionales, referidos como *bloques funcionales*. En algunos casos, iremos a través del proceso de diseño para obtener un circuito a partir de la función, mientras en otros casos, simplemente presentaremos la función y una implementación de ella. Estas funciones tienen una importancia especial en el diseño digital. En el pasado, los bloques funcionales se fabricaban como circuitos integrados de pequeña y media escala. Hoy, en circuitos de muy alta escala de integración (VLSI), los bloques funcionales se emplean para diseñar circuitos con muchos de estos bloques. Las funciones combinacionales y sus implementaciones son fundamentales para entender los circuitos VLSI. Normalmente, mediante el empleo de jerarquía, construimos circuitos como instancias de estas funciones o de los bloques funcionales asociados.

Los circuitos de alta y muy alta escala de integración son, en su mayoría, circuitos secuenciales como los descritos en la Sección 3.1 y estudiados al comienzo del Capítulo 6. Las funciones y los bloques funcionales que se discuten en este Capítulo 4 son combinacionales. Sin embargo, están a menudo combinados con elementos de almacenamiento para formar circuitos secuenciales como se muestra en la Figura 4-1. Las entradas al circuito combinacional pueden proceder tanto del entorno exterior como de los elementos de almacenamiento. Las salidas del circuito combinacional van tanto hacia el entorno exterior como hacia los elementos de almacenamiento. En capítulos posteriores emplearemos las funciones y bloques combinacionales aquí definidos, Capítulo 5, junto con elementos de almacenamiento, Capítulo 6, para formar circuitos secuenciales que realicen funciones muy útiles. Además, las funciones y bloques definidos en los Capítulo 4 y 5, servirán, en este y siguientes capítulos, como base para describir y entender tanto los circuitos combinacionales como los secuenciales mediante el empleo de lenguajes de descripción hardware.



□ **FIGURA 4-1**
Diagrama de bloques de un circuito secuencial

4-2 FUNCIONES LÓGICAS BÁSICAS

La asignación, transferencia, inversión y habilitación son algunas de las funciones lógicas combinacionales más elementales. Las dos primeras operaciones, la asignación y la transferencia, no implican ningún operador booleano. Sólo usan variables y constantes. Como consecuencia, en la implementación de estas operaciones no se emplean puertas lógicas. La inversión (o complemento) supone el uso de una única puerta por variable, y la habilitación implica el empleo de una o dos puertas lógicas por variable.

Asignación, transferencia y complemento

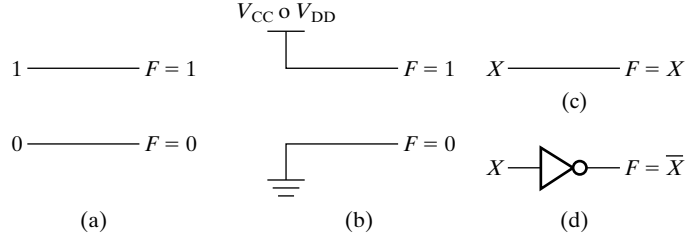
Si una función de un único bit depende de una única variable X, serán posibles, como mucho, cuatro funciones diferentes. La Tabla 4-1 muestra las tablas de verdad para estas funciones.

□ **TABLA 4-1**
Funciones de una variable

X	$F = 0$	$F = X$	$F = \overline{X}$	$F = 1$
0	0	0	1	1
1	0	1	0	1

La primera y la última de las columnas de la tabla asignan el valor constante 0 y el valor constante 1 a la función respectivamente, por consiguiente, llevan a cabo una *asignación*. En la segunda columna, la función es simplemente la variable de entrada X, de este modo X se *transfiere* de la entrada a la salida. En la tercera columna, la función es $\overline{X}$, por tanto la entrada X se *complementa* para convertirse en la salida $\overline{X}$.

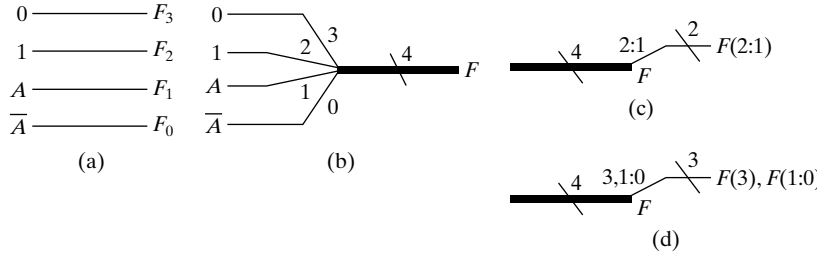
La Figura 4-2 muestra las implementaciones para estas cuatro funciones. La asignación de valores fijos se implementa conectando una constante 0 o 1 a la salida F, tal y como muestra la Figura 4-2(a). La Figura 4-2(b) muestra una representación alternativa que se emplea en los esquemas lógicos. Empleando lógica positiva, la constante 0 se representa por el símbolo de masa y la constante 1 por el símbolo de la tensión de alimentación. Este último símbolo puede nombrarse como V_{CC} o V_{DD}. La transferencia se implementa conectando un único cable desde X hacia F, tal y como se aprecia en la Figura 4-2(c). Finalmente, el complemento se representa mediante un inversor que logra $F = \overline{X}$ a partir de la entrada X como se observa en la Figura 4-2(d).



□ **FIGURA 4-2**
Implementación de funciones de una única variable X

Funciones de varios bit

Las funciones definidas con anterioridad pueden aplicarse a varios bits simultáneamente. Podemos pensar en estas funciones de múltiples bits como vectores de funciones de un único bit. Por ejemplo, supongamos que tenemos cuatro funciones, F_3, F_2, F_1 y F_0 que construyen una función F de 4 bits. Ordenaremos las cuatro funciones de modo que F_3 sea el bit más significativo y el F_0 el bit menos significativo, proporcionando el vector $F = (F_3, F_2, F_1, F_0)$. Suponga que F consiste en las siguientes funciones básicas $F_3 = 0$, $F_2 = 1$, $F_1 = A$ y $F_0 = \overline{A}$. Entonces podemos escribir F como el vector $(0, 1\ A, \overline{A})$. Para $A = 0$, $F = (0, 1, 0, 1)$ y para $A = 1$, $F = (0, 1, 1, 0)$. Esta función de múltiples bis puede ser referida como $F(3:0)$ o simplemente como F y su implementación se muestra en la Figura 4-3(a). Por comodidad, en los esquemas hemos representado un conjunto de varios hilos relacionados empleando una única línea de mayor grosor con una barra (*slash*) cruzándola. El entero que acompaña a la barra indica el número de cables, cómo muestra la Figura 4-3(b). Para poder conectar los valores 0, 1, X y $\overline{X}$ a los bits apropiados de F, separamos F en cuatro cables, uno por cada bit de F, y los nombramos adecuadamente.



□ **FIGURA 4-3**
Implementación de funciones básicas de varios bits

Del mismo modo, puede ocurrir en el proceso de transferencia, que sólo queramos usar un pequeño grupo de elementos de *F*, por ejemplo, F_2 y F_1. La Figura 4-3(c) muestra la notación utilizada con este fin para los bits de *F*. La Figura 4-3(d) se ha empleado para ilustrar un caso más complejo que utiliza F_3, F_1 y F_0. Observe que, puesto que, F_3, F_1 y F_0 no están juntos, no podemos usar la anotación *F*(3:0) para referirnos a este subvector. En su lugar emplearemos una combinación de dos su vectores, *F*(3), *F*(1:0) nombrado por los subíndices 3, 1:0. La notación real empleada para los vectores y subvectores varía dependiendo de las herramientas de captura de esquemas y HDL disponibles. La Figura 4-3 muestra una de estas propuestas. Para cada herramienta determinada, debe consultarse la documentación.

La asignación de valores, la transferencia y el complemento tienen una gran variedad de aplicaciones en los diseños lógicos. La asignación de valores implica sustituir una o más variables por los valores constantes 1 y 0. La asignación puede ser permanente o temporal. En la asignación permanente, el valor nunca se podrá modificar. En la asignación temporal, los valores pueden ser cambiados, empleando a menudo mecanismos algo diferentes a los empleados en las operaciones lógicas ordinarias. La asignación de valores permanentes o fijos tiene su aplicación principal en los dispositivos de lógica programable. Cualquier función que no esté contemplada en un dispositivo programable puede ser implementada fijando un conjunto de valores, como se muestra el próximo ejemplo.

EJEMPLO 4-1 Asignación de valores para implementar una función

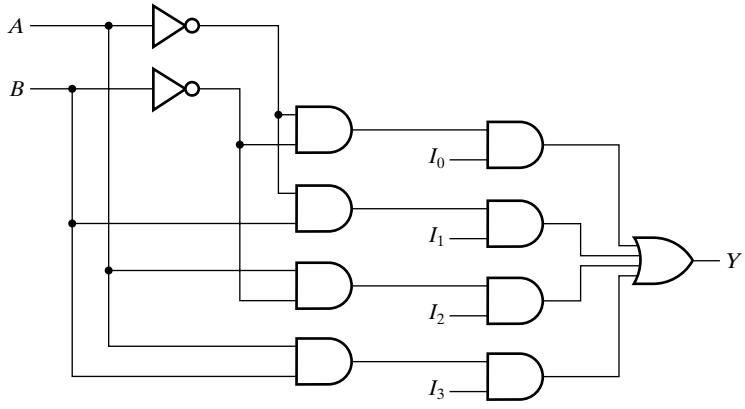
Considere la tabla de verdad mostrada en la Figura 4-4(a). *A* y *B* son dos variables de entrada, como también lo son desde I_0 hasta I_3. Dependiendo de la función que se desee, a las variables I_0, I_1, I_2 e I_3 se les asignaran valores de 0 o 1. Observe que *I* es, en realidad, una función de seis variables cuya tabla de verdad expandida está formada por 64 filas y 7 columnas. Pero, al colocar I_0, I_1, I_2 e I_3 en la columna de salida, hemos reducido considerablemente el tamaño de la tabla. La ecuación para la salida *Y* de esta tabla es:

$$Y(A, B, I_0, I_1, I_2, I_3) = \bar{A}\bar{B}I_0 + \bar{A}BI_1 + A\bar{B}I_2 + ABI_3$$

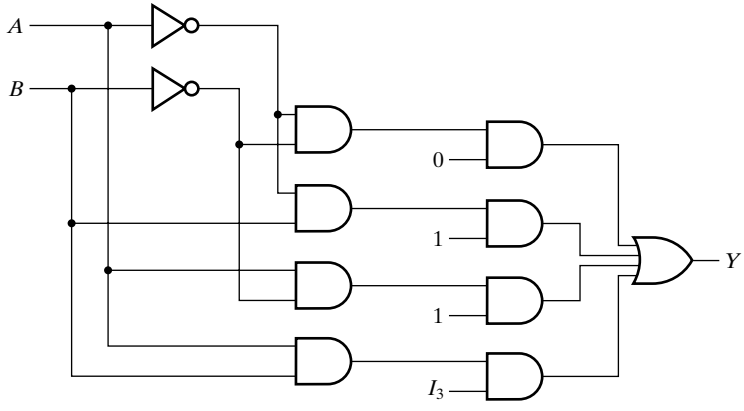
La Figura 4-4(b), muestra la implementación para esta ecuación. Al fijar los valores desde I_0 hasta I_3, se puede implementar cualquier función *Y*(*A*, *B*). Como muestra la Tabla 4-2, $Y = A + B$ puede implementarse usando $I_0 = 0$, $I_1 = 1$, $I_2 = 1$ e $I_3 = 1$. O podemos implementar $Y = A\bar{B} + \bar{B}A$ usando $I_0 = 0$, $I_1 = 1$, $I_2 = 1$ e $I_3 = 0$. Cualquiera de estas funciones puede implementarse permanente o temporalmente sin más que fijar $I_0 = 0$, $I_1 = 1$, $I_2 = 1$, y usando I_3 como una variable que toma el valor 1 para $A + B$ y el valor 0 para $AB + \bar{A}\bar{B}$. La Figura 4-4(c) muestra el circuito final.

A	B	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(a)



(b)



(c)

□ **FIGURA 4-4**
Implementación de dos funciones usando asignación de valores

□ **TABLA 4-2**
Implementación de una función por asignación de valores

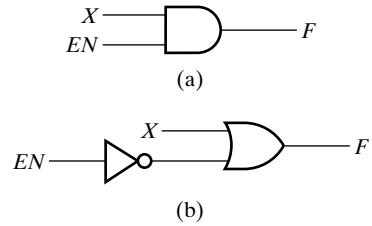
A	B	$Y = A + B$	$Y = A\bar{B} + \bar{A}B$	$Y = A + B(I_3 = 1)$ o $Y = A\bar{B} + \bar{A}B(I_3 = 0)$
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	I_3

■

Habilitación

El concepto de habilitar una señal apareció por primera vez en la Sección 2-9 donde se introdujeron los conceptos de salida en alta impedancia (*Hi-Z*) y *buffer* tri-estado. En general, la habilitación permite que una señal de entrada pase hacia la salida. Además de reemplazar la señal de entrada por un estado de alta impedancia en la salida, la deshabilitación también puede sustituir

la señal de entrada por un valor fijo en la salida, bien sea 0 o 1. La señal de entrada adicional, a menudo denominada *ENABLE* o *EN*, es necesaria para determinar cuándo la salida está habilitada o no lo está. Por ejemplo, si la señal *EN* tiene un 1, la entrada *X* pasará directamente a la salida (habilitada) pero si la señal *EN* está a 0, la salida mostrará un 0 fijo (deshabilitada). En estos casos, si el valor deshabilitado está a 0 fijo, la señal de entrada *X* es multiplicada (AND) a la señal *EN*, como muestra la Figura 4-5(a). Si el valor deshabilitado está a 1, entonces la señal de entrada *X* será sumada (OR) con el complemento de la señal *EN*, tal y como muestra la Figura 4-5(b). Por otra parte, la señal de salida puede ser habilitada con $EN = 0$ en lugar de 1, pasando a denominarse entonces $\overline{EN}$, pues *EN* se ha invertido, como en la Figura 4-5(b).



□ **FIGURA 4-5**
Circuitos habilitadores

EJEMPLO 4-2 Aplicación de habilitación

En la mayoría de los automóviles, la luz, la radio y las ventanilla sólo funcionan si el interruptor del contacto está encendido. En este caso, el contacto actúa como una señal «habilitadora». Suponga que queremos copiar este sistema automovilístico usando las siguientes variables y definiciones:

Contacto *CN* - A 0 si está abierto, a 1 cerrado
Interruptor de luz *IL* - A 0 si está abierto, a 1 cerrado
Interruptor de radio *IR* - A 0 si está abierto, a 1 cerrado
Interruptor de las ventanillas *IV* - A 0 si está abierto, a 1 cerrado
Luces *L* - A 0 apagadas, a 1 encendidas
Radio *R* - A 0 apagada, a 1 encendida
Alimentación de las ventanillas *V* - A 0 desconectada, a 1 conectada

La Tabla 4-3 contiene la tabla de verdad resumida para las operaciones de este sistema. Observe cómo, cuando el interruptor de arranque está abierto $CN = 0$, todos los accesorios que controla están apagados (0) a pesar del estado en que se encuentren sus interruptores.

Esto se refleja en la primera fila de la tabla. Con el empleo de las indiferencias (X), esta tabla resumida de tan sólo nueve filas representa la misma información que la habitual tabla de verdad de 16 filas. Mientras las X en las columnas de salida representan condiciones indiferentes, las X en las columnas de entrada se usan para representar productos de términos que no son mini términos. Por ejemplo 0XXX representa el producto $\overline{CN}$. En los mini términos, cada variable es negada si el bit correspondiente en la combinación es 0 y no será negada si el bit es 1. Si el bit correspondiente en la combinación de entrada es X, entonces la variable no aparecerá en el producto de términos. Cuando el interruptor del contacto está encendido $CN = 1$, entonces los accesorios se controlarán por sus respectivos interruptores. Cuando *CN* está apagado (0), todos los accesorios también estarán apagados. Entonces *CN* modifica los valores normales de las salidas *L*, *R* y *V* por un valor 0 fijo, dando sentido a la definición de una señal *ENABLE*.

□ **TABLA 4-3**
Tabla de verdad de una aplicación de habilitación

Interruptores de entrada				**Controles**		
CN	***IL***	***IR***	***IV***	***L***	***R***	***V***
0	X	X	X	0	0	0
1	0	0	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	1	0	0
1	1	0	1	1	0	1
1	1	1	0	1	1	0
1	1	1	1	1	1	1

■

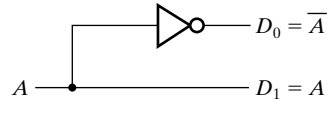
4-3 DECODIFICACIÓN

En las computadora digitales, se emplean códigos binarios para representar cantidades discretas de información. Un código binario de n bits es capaz de representar hasta 2^n elementos distintos de información codificada. Decodificar es convertir un código de entrada de n bits en un código de salida de m bits, $n \leqslant m \leqslant 2^n$, tal que para cada palabra válida codificada en la entrada exista un único código de salida. La decodificación es realizada por un *decodificador*, un circuito combinacional al que se aplica un código binario de n bits por sus entradas y genera un código binario de m bits por sus salidas. Puede ocurrir que para ciertas combinaciones de entrada no usadas el decodificador no genera ningún código por las salidas. Entre todas las funciones especializadas definidas aquí, la decodificación es la más importante y sus correspondientes bloques funcionales se incorporan en muchas de otras funciones y en bloques funcionales definidos aquí.

En esta sección, los bloques funcionales que implementan la decodificación se denominan decodificadores de n a m líneas, donde $m \leqslant 2^n$. Su propósito es generar 2^n (o menos) mini términos a partir de las n variables de entrada. Para $n = 1$ y $m = 2$ obtenemos la función decodificadora de 1 a 2 líneas con una entrada A y salidas D_0 y D_1. La Figura 4-6(a) muestra la tabla de verdad para esta función decodificadora. Si $A = 0$, entonces $D_0 = 1$ y $D_1 = 0$. Si $A = 1$, entonces $D_0 = 0$ y $D_1 = 1$. A partir de esta tabla de verdad, se obtiene $D_0 = \overline{A}$ y $D_1 = A$ dando el circuito que aparece en la Figura 4-6(b).

A	D_0	D_1
0	1	0
1	0	1

(a)



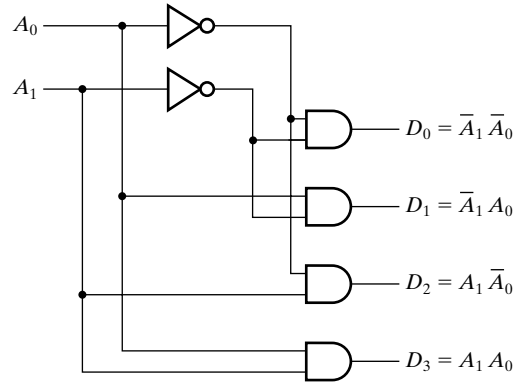
(b)

□ **FIGURA 4-6**
Decodificador de 1 a 2 líneas

En la Figura 4-7(a) se muestra la tabla de verdad de una segunda función decodificadora para $n = 2$ y $m = 4$, que ilustra mejor la naturaleza general de los decodificadores. Las salidas de esta tabla son mini términos de dos variables, en cada fila aparece un valor de salida igual a 1 y 3 valores de salida iguales a 0. La salida D_i es igual a 1 siempre que los dos valores de entrada A_1 y A_0 representen el código binario para el número i. Como consecuencia, el circuito implementa cuatro posibles mini términos de dos variables, un mini término para cada salida. En el diagrama lógico de la Figura 4-7(b), cada mini término se implementa mediante una puerta AND de 2 entradas. Estas puertas AND están conectadas a dos decodificadores de 1 a 2 líneas, uno por cada una de las líneas conectadas a las entradas de la puerta AND.

A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)



(b)

□ **FIGURA 4-7**
Decodificador de 2 a 4 líneas

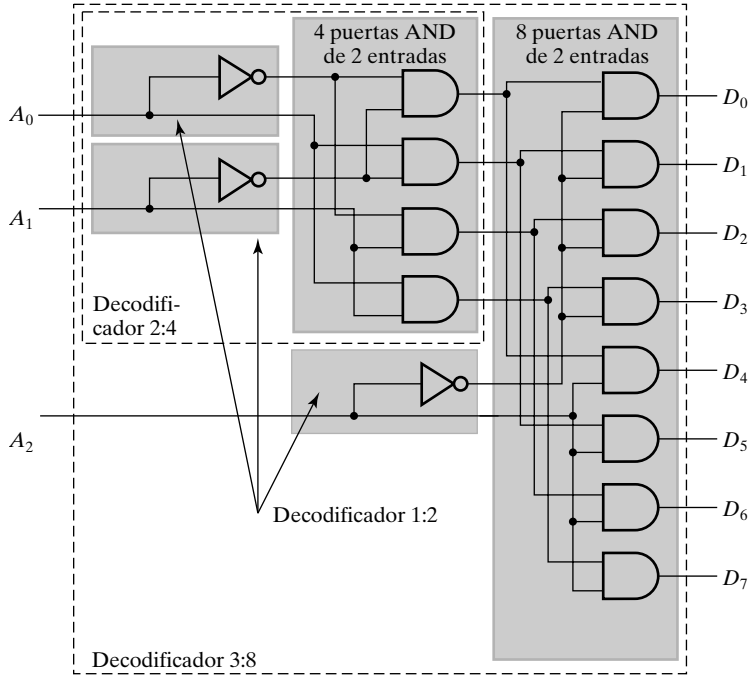
Extensión de decodificadores

Pueden construirse decodificadores mayores implementando simplemente cada función de mini términos con una puerta AND que tenga más entradas. Desafortunadamente, a medida que los decodificadores se hacen más grandes, esta solución da como resultado puertas con un elevado número de entradas. En esta sección, se proporciona un método que emplea diseño jerárquico y agrupaciones de puertas AND para construir cualquier decodificador con n entradas y 2^n salidas. El decodificador que se obtiene por este modo tiene el mismo o menor número total de entradas que el construido mediante la mera ampliación de cada puerta AND.

Para construir un decodificador de 3 a 8 líneas ($n = 3$) emplearemos, para formar los mini términos, un decodificador de 2 a 4 líneas y un decodificador de 1 a 2 líneas que alimentarán a 8 puertas AND de 2 entradas. Jerárquicamente, el decodificador de 2 a 4 puede implementarse usando decodificadores de 1 a 2 líneas que alimentan a 4 puertas AND de 2 entradas, como se observa en la Figura 4-7. La estructura resultante se muestra en la Figura 4-8.

El procedimiento general es el siguiente:

1. Hacer $k = n$.
2. Si k es par, dividir k entre 2 para obtener $k/2$. Emplear 2^k puertas AND conectadas a dos decodificadores de tamaño de salida $2^{k/2}$. Si k es impar, calcular $(k + 1)/2$ y $(k - 1)/2$. Usar 2^k puertas AND conectadas a un decodificador de tamaño de salida $2^{(k+1)/2}$ y un decodificador de tamaño de salida $2^{(k-1)/2}$.



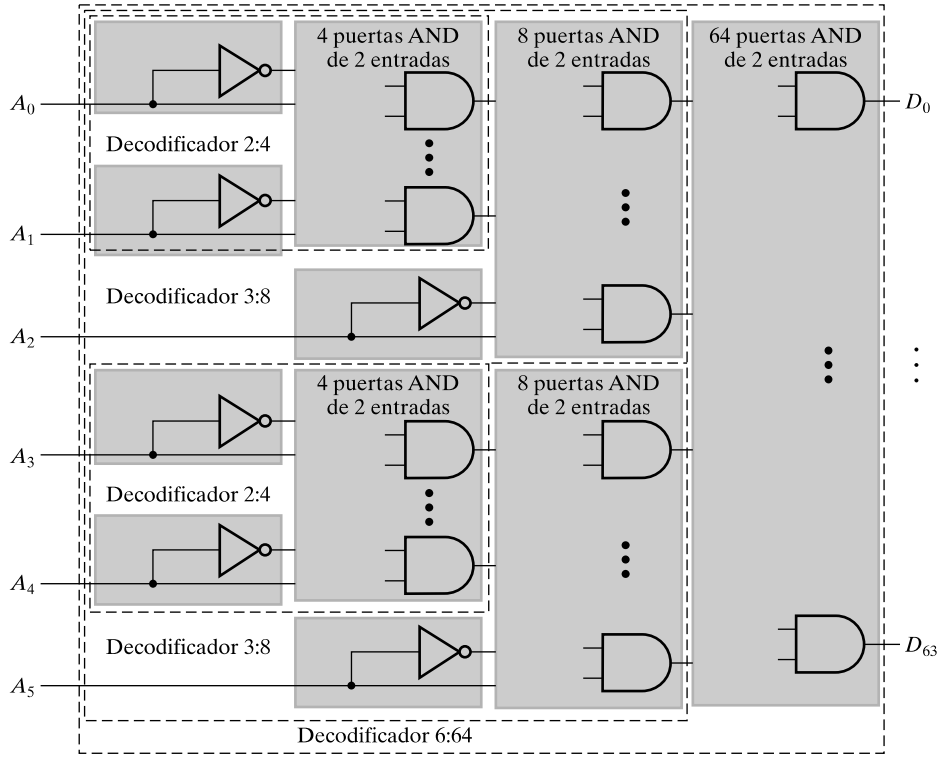
□ **FIGURA 4-8**
Decodificador de 3 a 8 líneas

3. Para cada uno de los decodificadores resultantes en el paso 2, repita el paso 2 desde k igual a los valores obtenidos en el paso 2 hasta $k = 1$. Para $k = 1$, use un decodificador de 1 a 2 líneas.

EJEMPLO 4-3 Decodificador de 6 a 64 líneas

Para un decodificador de 6 a 64 líneas ($k = n = 6$), en la primera ejecución del paso 2, se conectan 64 puertas AND de 2 entradas a dos decodificadores de tamaño de salida $2^3 = 8$ (es decir, por dos decodificadores de 3 a 8 líneas). En la segunda ejecución del paso 2, es $k = 3$. Puesto que k es impar, el resultado es que $(k + 1)/2 = 2$ y $(k - 1)/2 = 1$. Se conectan 8 puertas AND de 2 entradas a un decodificador de tamaño de salida $2^2 = 4$ y a un decodificador de tamaño de salida (es decir, a dos decodificadores de 2 a 4 líneas y a un decodificador de 2 a 1 líneas, respectivamente). Finalmente, en la siguiente ejecución del paso 2, $k = 2$, dando lugar a cuatro puertas AND de 2 entradas conectadas a dos decodificadores con tamaño de salida 2 (es decir, a dos decodificadores de 1 a 2 líneas). Puesto que todos los decodificadores se han expandido, el algoritmo del paso 3 termina en este momento. La Figura 4-9 muestra la estructura resultante. Esta estructura tiene un número total de entradas de $6 + 2(2 \times 4) + 2(2 \times 8) + 2 \times 64 = 182$. Si se hubiera empleado una única puerta AND para cada mini término, el número total de entradas habría sido $6 + (6 \times 64) = 390$, con lo que se ha conseguido una reducción significativa en el número total de entradas. ■

Como alternativa, suponga que se necesitan múltiples decodificadores y que éstos tienen variables de entrada comunes. En este caso, en lugar de implementar decodificadores distintos



□ **FIGURA 4-9**
Decodificador de 6 a 64 líneas

algunas partes de ellos podrán compartirse. Por ejemplo, suponga que 3 decodificadores d_a, d_b y d_c son función de las siguientes variables de entrada:

$$d_a(A, B, C, D)$$
$$d_b(A, B, C, E)$$
$$d_c(C, D, E, F)$$

d_a y d_b pueden compartir un decodificador de 3 a 8 líneas para A, B y C. d_a y d_c pueden compartir un decodificador de 2 a 4 líneas para C y D. d_b y d_c pueden compartir un decodificador 2 a 4 líneas para C y E. Al implementar estos tres bloques compartidos, C aparecería en los tres decodificadores diferentes y el circuito presentaría redundancia. Para usar C solamente una vez en los decodificadores compartidos, tendremos en cuenta los siguientes casos:

1. (A, B) compartido por d_a y d_b, y (C, D) compartido por d_a y d_c
2. (A, B) compartido por d_a y d_b, y (C, E) compartido por d_b y d_c, o
3. (A, B, C) compartido por d_a y d_b

Puesto que los casos 1 y 2 tienen claramente el mismo coste, compararemos el coste de los casos 1 y 3. Para el caso 1, el coste de las funciones d_a, d_b y d_c se reducirá en el coste de 2 decodificadores de 2 a 4 líneas (excepto los inversores) o 16 entradas de puertas. Para el caso 3, el coste para las funciones d_a y d_b se reduce en un decodificador de 3 a 8 líneas, o 24

entradas. Por tanto, es el caso 3 el que debe implementarse. La implementación formal de este algoritmo va más allá de nuestras posibilidades, solo se muestra un dibujo esquemático de esta aproximación.

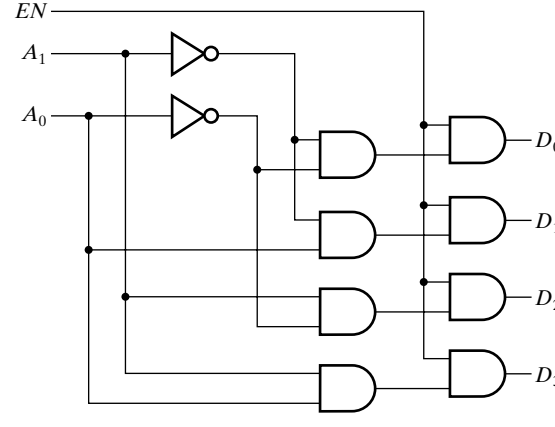
Decodificadores con señal de habilitación

La función decodificador de n a m líneas con habilitación puede implementarse conectando m circuitos habilitadores a las salidas del decodificador. De este modo, m copias de la señal habilitadora EN se conectarán a la entrada del control de habilitación de los circuitos habilitadores. Para $n = 2$ y $m = 4$, resulta el decodificador 2 a 4 líneas y con señal de habilitación mostrado en la Figura 4-10, junto a su tabla de verdad. Para $EN = 0$, todas las salidas del decodificador son 0. Para $EN = 1$, sólo una de las salidas del decodificador, determinada por el valor de (A_1, A_0), es 1 y todas las demás son 0. Si el decodificador controla un conjunto de luces, cuando la señal $EN = 0$, las luces estarán apagadas, y cuando $EN = 1$, solamente una luz estará encendida, con las otras tres apagadas. Para decodificadores mayores ($n \geqslant 4$), el número total de entradas puede reducirse colocando los circuitos habilitadores en las entradas del decodificador y sus negadas, en vez de en cada una de las salidas del decodificador.

En la Sección 4-5, se tratará la selección mediante el empleo de multiplexores. Lo contrario a la selección es la *distribución*, en la cual la información recibida procedente de una única línea es transmitida a cada una de las 2^n posibles líneas de salida. El circuito que implementa esta distribución se denomina *demultiplexor*. Para controlar qué señal de entrada es transmitida a la salida, se emplea una combinación de bits sobre las n líneas de selección. El decodificador 2 a 4 líneas con habilitación de la Figura 4-10 es una implementación de un demultiplexor de 1 a 4 líneas. En el demultiplexor, la entrada EN proporciona los datos, mientras que las otras entradas actúan como variables de selección. Aunque los dos circuitos tienen aplicaciones diferentes, sus diagramas lógicos son exactamente los mismos. Por esta razón, un decodificador con entrada de habilitación se denomina también decodificador/demultiplexor. La entrada de datos EN tiene conexión hacia las cuatro salidas, pero la información de entrada sólo es direccionada hacía una de ellas, especificada mediante las dos líneas de selección A_1 y A_0. Por ejemplo, si $(A_1, A_0) = 10$, la salida D_2 tiene el valor aplicado en la entrada EN, mientras las demás salidas

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(a)



(b)

□ **FIGURA 4-10**
Decodificador con señal de habilitación de 2 a 4 líneas

permanecen inactivas mostrando un 0 lógico. Si el decodificador controla un conjunto de cuatro luces, con (A_1, A_0) = 10 y *EN* alternativamente cambiando entre 1 y 0, la luz controlada por D_2 lucirá intermitentemente, mientras que todas las demás luces estarán apagadas.

4-4 CODIFICACIÓN

Un codificador es una función digital que realiza la operación inversa del decodificador. Un codificador tiene 2^n (o menos) líneas de entrada y n líneas de salida. Las líneas de salida generan el código binario correspondiente a los valores de la entrada. Un ejemplo de codificador es el codificador de Octal a Binario que se muestra en la Tabla 4-4. Este codificador tiene 8 entradas, una por cada uno de los dígitos que soporta, y 3 salidas que generan el correspondiente número binario. Si suponemos que sólo una de las entradas puede tomar el valor 1 al mismo tiempo, entonces la tabla sólo tendrá ocho filas, cada una con los valores de salida especificados. Para las 56 combinaciones restantes, todas las salidas serán indiferencias. A partir de la tabla de verdad observamos como A_i es 1 para las columnas en las que D_j es 1 cuando el subíndice j se representa en binario con un 1 en la posición i. Por ejemplo, la salida es $A_0 = 1$ si la entrada es 1, 3, 5 o 7. Puesto que todos estos valores son impares, tienen un 1 en la posición 0 de su correspondiente representación binaria. Esta aproximación se puede emplear para encontrar la tabla de verdad. A partir de la tabla, el decodificador puede implementarse con n puertas OR, una por cada una de las variables A_i de salida. Cada puerta OR combina las variables de entrada D_j de las filas que tienen un 1 para un valor $A_i = 1$. Para el codificador de 8 a 3 líneas, las ecuaciones de salida resultantes son:

$$A_0 = D_1 + D_3 + D_5 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$

que se implementan con 3 puertas OR de 4 entradas.

El codificador recién descrito presenta la limitación de que sólo una de las entradas puede estar activa al mismo tiempo: si dos entradas se activan simultáneamente, la salida presenta una

□ **TABLA 4-4**
Tabla de verdad para un codificador octal a binario

Entradas								Salidas		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

combinación incorrecta. Por ejemplo, si D_3 y D_6 son puestas simultáneamente a 1, la salida del codificador será 111 puesto que las tres salidas son iguales a 1. Esta combinación no es la representación binaria ni del 3 ni del 6. Para resolver esta ambigüedad, algunos circuitos codificadores establecen una prioridad en las entradas para asegurar que sólo una de ellas sea codificada. Si asignamos una prioridad mayor a las entradas que representan números mayores, y si tanto D_3 como D_6 se han colocado a 1 al mismo tiempo, la salida será 110 porque D_6 tiene mayor prioridad que D_3. En el codificador octal a binario se produce otra ambigüedad cuando todas las entradas están a 0, al mostrar ceros en todas las salidas, igual que se produce cuando la entrada D_0 es 1. Esta discrepancia puede resolverse proporcionando una salida adicional que indique que una de las entradas, al menos, se ha puesto a 1.

Codificador con prioridad

Un codificador con prioridad es un circuito combinacional que implementa una función de prioridad. Como se ha mencionado en el párrafo anterior, la función del codificador con prioridad es tal que si dos o más entradas son iguales a 1 al mismo tiempo, aquella que tenga mayor prioridad tomará la delantera. La tabla de verdad para un codificador de prioridad de cuatro entradas se da en la Tabla 4-5. Con el empleo de las X, esta tabla reducida de cinco filas representa la misma información que la tabla de verdad habitual de 16 filas. Mientras que las X en las columnas de salida representan condiciones indiferentes, las X en las columnas de entrada se usan para representar productos de términos que no son mini términos. Por ejemplo 001X representa el producto de términos $\bar{D}_3\bar{D}_2D_1$. Tal y como ocurre con los mini términos, cada variable es negada si el bit correspondiente en la combinación de entrada de la tabla es 0 y no es invertida si el bit es 1. Si el correspondiente bit en la combinación de entrada es una X, entonces la variable no aparece en el producto de términos. De este modo, para 001X, la variable D_0, correspondiente a la posición de la X, no aparecerá en $\bar{D}_3\bar{D}_2D_1$.

El número de filas de una tabla de verdad completa representadas por una única fila en la tabla resumida es 2^p, donde p es el número de X en la fila. Por ejemplo, en la Tabla 4-5, la fila 1XXX representa $2^3 = 8$ filas de la tabla de verdad completa que tienen el mismo valor para todas las salidas. Para construir una tabla de verdad resumida debemos incluir cada mini término en al menos una de las filas, en el sentido de que cada mini término puede obtenerse reemplazando por 1 y 0 las X. Del mismo modo, un mini término nunca debe incluirse en más de una fila, de modo que no existan conflictos entre las salidas de varias filas.

Formamos la Tabla 4-5 como sigue: la entrada D_3 es la de mayor prioridad; por tanto no tendremos en cuenta los valores de las otras entradas cuando esta entrada esté a 1, la salida para

□ **TABLA 4-5**
Tabla de verdad de un codificador con prioridad

Entradas				Salidas		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

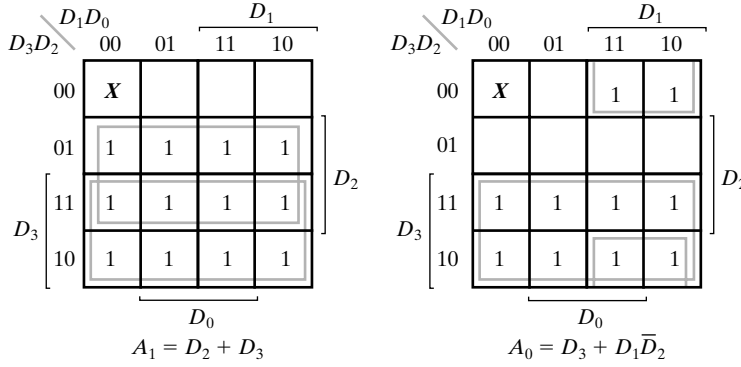
A_1A_0 es 1 (el binario de 3). A partir de aquí hemos obtenido la última fila de la Tabla 4-5. D_2 tiene el siguiente nivel de prioridad. La salida es 10 si D_2 es 1, y siempre que D_3 sea 0, no teniendo en cuenta los valores de las entradas de menor prioridad. De este modo, obtenemos la cuarta fila de la tabla. La salida para D_1 se genera sólo si todas las entradas con mayor prioridad están a 0, y sin tener en cuenta los niveles de prioridad que estén por debajo. De este modo, obtenemos las restantes filas de la tabla. La salida de validación designada como V es puesta a 1 sólo cuando una o más de las entradas son iguales a 1. Si todas las entradas son 0, entonces $V = 0$ y las otras dos salidas del circuito no se emplearán, siendo referidas como indiferencias en la parte de la tabla destinada a las salidas.

Los mapas para simplificar las salidas A_1 y A_0 se muestran en la Figura 4-11. A partir de la Tabla 4-5 se han obtenido los mini términos de las dos funciones. Los valores de salida de la tabla se han transferido directamente a los mapas situándolos en los cuadros cubiertos por el correspondiente producto. La ecuación optimizada para cada función se ha colocado debajo de su mapa correspondiente. La ecuación para la salida V es una función OR de todas las variables de entrada. El codificador con prioridad se ha implementado en la Figura 4-12 de acuerdo con las siguientes funciones booleanas:

$$A_0 = D_3 + D_1\bar{D}_2$$

$$A_1 = D_2 + D_3$$

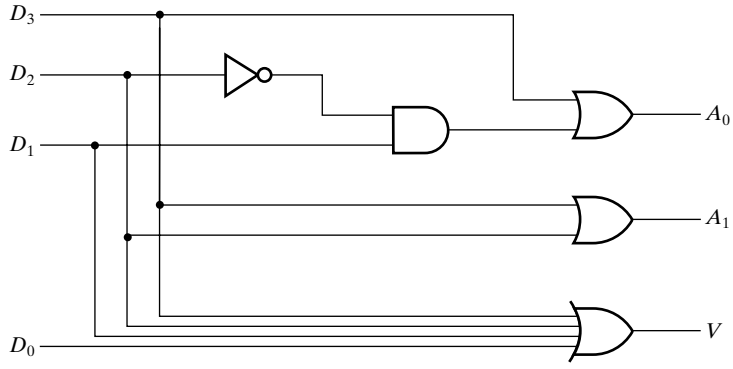
$$V = D_0 + D_1 + D_2 + D_3$$



□ **FIGURA 4-11**
Mapas para el codificador con prioridad

Expansión de codificadores

Hasta ahora, sólo hemos tenido en cuenta codificadores pequeños. Los codificadores pueden ampliarse para un mayor número de entradas mediante la expansión de puertas OR. En la implementación de codificadores, el empleo de circuitos de múltiples niveles para las puertas OR, compartidos para la obtención de los bits más significativos del código de salida, reduce el número total de entradas cuando $n \geqslant 5$. Para $n \geqslant 3$ el mapeado tecnológico, debido a las limitaciones en el *fan-in* de las puertas, resulta en la generación directa de estos circuitos de múltiples niveles.



□ **FIGURA 4-12**
Diagrama lógico para un codificador con prioridad de 4 entradas

4-5 SELECCIÓN

En una computadora, la selección de información es una función muy importante, no sólo en la comunicación entre las partes del sistema, sino también dentro de las propias partes. En otras aplicaciones la selección, en combinación con la asignación de valores, permite implementar funciones combinacionales. Normalmente, los circuitos que llevan a cabo la selección se componen de una serie de entradas de entre las que se realiza la selección, una única salida y un conjunto de líneas de control para determinar la selección a realizar. Primero consideraremos la selección usando multiplexores; más tarde examinaremos, brevemente, los circuitos de selección implementados con puertas tri-estado y puertas de transmisión.

Multiplexores

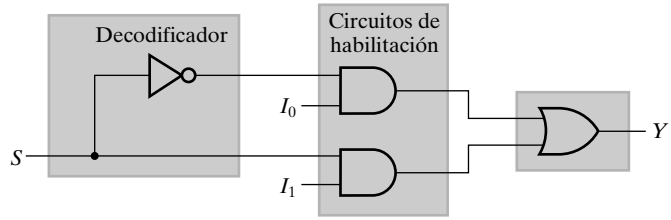
Un multiplexor es un circuito combinacional capaz de seleccionar una información binaria procedente de una de entre varias líneas de entrada y direccionar dicha información hacia una única línea de salida. La selección de una línea de entrada en particular se controla mediante un conjunto de variables de entrada, denominadas *líneas de selección*. Normalmente hay 2^n líneas de entrada y n entradas de selección, cuya combinación de bits determina qué entrada será seleccionada. Comenzamos con $n = 1$, un multiplexor de 2 a 1. Esta función tiene dos entradas de información, I_0 e I_1, y una única entrada de selección S. La tabla de verdad para este circuito es la mostrada en la Tabla 4-6. Examinando dicha tabla, si la entrada de selección es $S = 1$, la salida del multiplexor tomará el valor de I_1 y si la entrada de selección es $S = 0$, entonces la salida del multiplexor tomará el valor de I_0. De este modo, S selecciona cuál de las entradas I_0 o I_1 aparece en la salida Y. A partir de esta discusión, podemos ver que la ecuación para la salida del multiplexor 2 a 1 Y es:

$$Y = \bar{S}I_0 + SI_1$$

Esta misma ecuación puede obtenerse utilizando un Mapa de Karnaugh de 3 variables. Tal y como muestra la Figura 4-13, la implementación de la ecuación anterior puede descomponerse en un decodificador de 1 a 2 líneas, dos circuitos de habilitación y una puerta OR de 2 entradas.

□ **TABLA 4-6**
Tabla de verdad de un multiplexor 2 a 1

S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



□ **FIGURA 4-13**
Multiplexor de un solo bit 2 a 1

Suponga que deseamos diseñar un multiplexor de 4 a 1 líneas. En este caso, la función Y depende de cuatro entradas I_0, I_1, I_2 e I_3 y dos entradas de selección S_0 y S_1. Colocando en la columna Y los valores desde I_0 hasta I_3, podemos construir la Tabla 4-7, una tabla de verdad resumida para este multiplexor. En esta tabla, la información de las variables de entrada no aparece en las columnas de entrada de la tabla pero aparece en la columna de salida. Cada fila de la tabla resumida representa muchas filas de la tabla de verdad completa. En la Tabla 4-7, la fila $00I_0$ representa todas filas en las cuales $(S_1, S_0) = 00$, para $I_0 = 1$ da $Y = 1$ y para $I_0 = 0$ da $Y = 0$. Puesto que hay seis variables y sólo S_1 y S_0 son fijos, una única fila representa o equivale a 16 filas de la correspondiente tabla de verdad completa. A partir de esta tabla, podemos escribir la ecuación para Y como:

$$Y = \bar{S}_1\bar{S}_0I_0 + \bar{S}_1S_0I_1 + S_1\bar{S}_0I_2 + S_1S_0I_3$$

□ **TABLA 4-7**
Tabla de verdad resumida de un multiplexor 4 a 1

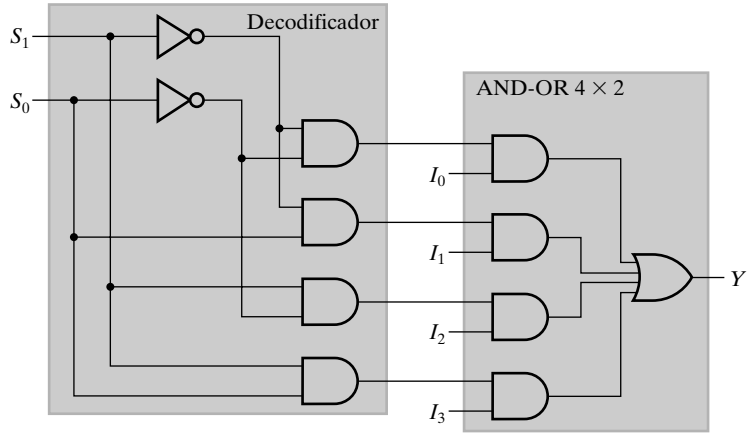
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Si esta ecuación se implementara directamente, necesitaríamos dos inversores, cuatro puertas AND de 3 entradas y una puerta OR de cuatro entradas, dando un número total de entradas de 18. Puede obtenerse una implementación diferente factorizando los términos AND, de modo que quedaría:

$$Y = (\bar{S}_1\bar{S}_0)I_0 + (\bar{S}_1S_0)I_1 + (S_1\bar{S}_0)I_2 + (S_1S_0)I_3$$

Esta implementación puede construirse combinando un decodificador 2-4 líneas, cuatro puertas AND empleadas como circuitos de habilitación y una puerta OR de 4 entradas, tal y como muestra la Figura 4-14. Nos referiremos a la combinación de puertas AND y OR como $m \times 2$ AND-OR, donde m es el número de puertas AND y 2 es el número de entradas de las puertas AND. El circuito resultante cuenta con 22 entradas de puertas, lo que incrementa su coste. Sin embargo, éste es la base estructural para la construcción, por expansión, de grandes multiplexores de n a 2^n líneas.

Un multiplexor también se denomina *selector de datos*, puesto que selecciona solo una de entre las muchas informaciones de entrada y lleva la información binaria hacia la línea de salida. El término «multiplexor» es a menudo abreviado por «MUX».



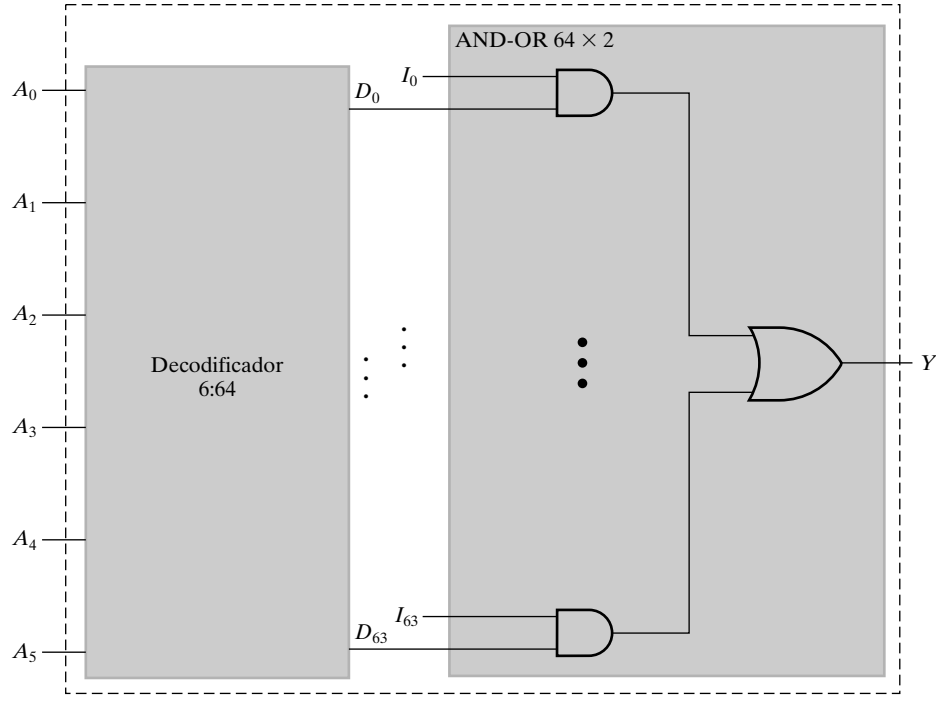
□ **FIGURA 4-14**
Multiplexor 4 a 1 de un solo bit

Expansión de multiplexores

Los multiplexores pueden expandirse considerando n vectores de bits de entrada. La expansión se basa en el empleo de circuitos cuya estructura viene dada por la Figura 4-14, consistiendo en un decodificador, circuitos habilitadores y puertas AND y OR. El diseño de multiplexores se ilustra en los Ejemplos 4-4 y 4-5.

EJEMPLO 4-4 Multiplexor de 64 a 1

Debe diseñarse un multiplexor para $n = 6$. Para ello, necesitaremos el decodificador de 6 a 64 líneas que se muestra en la Figura 4-9 y una puerta AND-OR. La estructura resultante se muestra en la Figura 4-15. Esta estructura presenta un número total de entradas $182 + 128 + 64 = 374$. En contraposición, si el decodificador y el circuito habilitador se reemplazan por inversores más puertas AND de 7 entradas, el número total de entradas necesario es $6 + 448 + 64 = 518$. Para



□ **FIGURA 4-15**
Multiplexor de 64 a 1 líneas

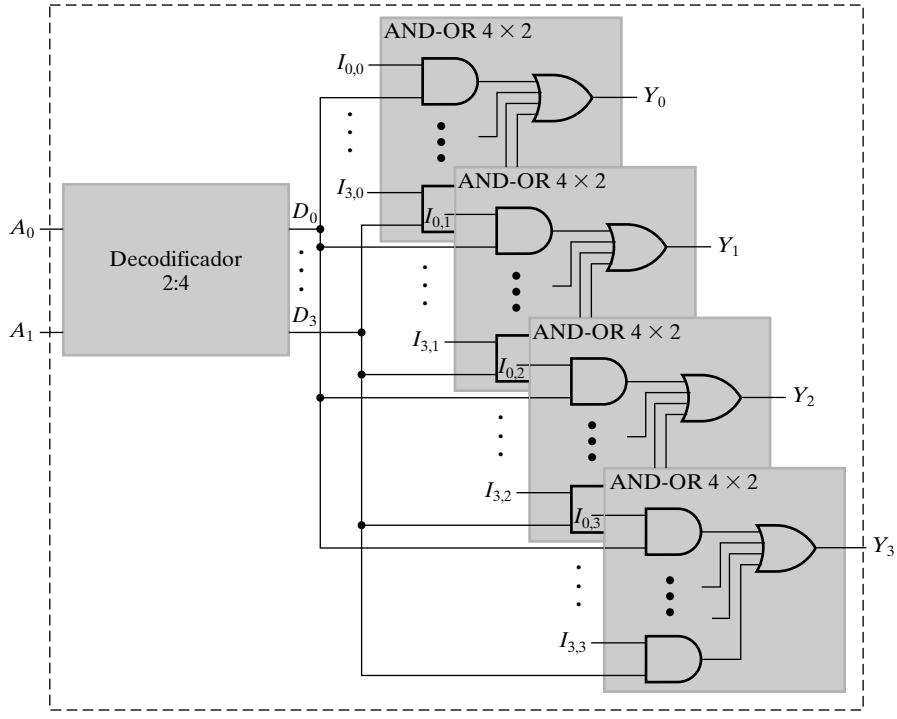
multiplexores de un único bit como éste, la combinación de las puertas AND que generan D_i con la puerta AND atacada por D_i en una única puerta AND de 3 entradas, para cada $i = 0$ hasta 63, reduce el número total de entradas a 310. Para multiplexores de múltiples bits, esta reducción a puertas AND de 3 entradas no puede llevarse a cabo sin reproducir exactamente las salidas AND de los decodificadores. Como resultado, en la mayoría de los casos la estructura original presenta una menor coste en número total de entradas. El siguiente ejemplo ilustra la expansión para un multiplexor múltiple. ■

EJEMPLO 4-5 Multiplexor cuádruple 4 a 1

Se va a diseñar un multiplexor cuádruple 4 a 1 con dos entradas de selección y donde cada entrada de información se ha sustituido por un vector de 4 entradas. Puesto que la información de entrada es un vector, la salida Y también será un vector de cuatro elementos. Para implementar este multiplexor se necesitan decodificadores de 2 a 4 líneas como los dados en la Figura 4-7, y cuatro puertas 4 × 2 AND-OR. La estructura resultante se muestra en la Figura 4-16. Esta estructura presenta un coste de 10 + 32 + 16 = 58 entradas de puertas. Sin embargo cuatro multiplexores de 4 entradas, implementados con puertas de 3 entradas, presentan un coste de 76 entradas de puertas. Por tanto compartiendo los decodificadores se reduce el coste. ■

Implementaciones alternativas de selectores

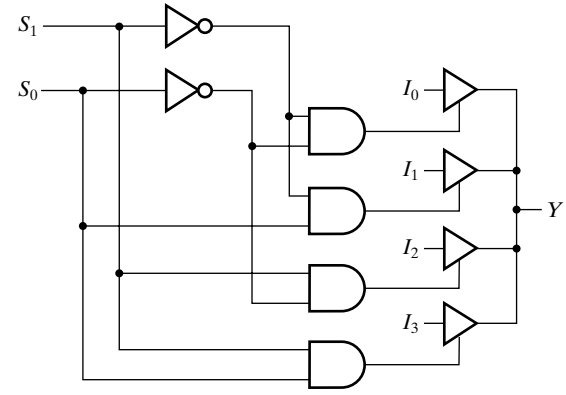
Es posible implementar selectores de datos y multiplexores empleando drivers tri-estado y puertas de transmisión, consiguiendo disminuir el coste requerido con puertas.



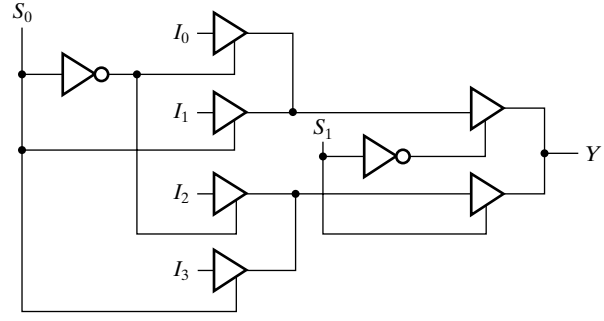
□ **FIGURA 4-16**
Cuádruple multiplexor de 4 a 1

IMPLEMENTACIONES CON TRI-ESTADO Los drivers triestado, introducidos en el Capítulo 2, proporcionan una implementación alternativa a los multiplexores. En la implementación que se muestra en la Figura 4-17(a), cuatro drivers tri-estado con sus salidas conectadas a *Y* sustituyen a los circuitos de habilitación y la puerta OR de salida, dando un número total de entradas de 18. Además, la lógica puede reducirse distribuyendo la decodificación a través de los drivers tri-estado, tal y cómo muestra la Figura 4-17(b). En este caso, tres pares de circuitos de habilitación, todos con decodificares de 2 salidas consistentes en un sencillo cable y un inversor, atacan a las entradas de habilitación. El número total de entradas para este circuito se reduce a sólo 14.

IMPLEMENTACIÓN CON PUERTAS DE TRANSMISIÓN Una modificación a la aproximación con puertas tri-estado de la Figura 4-17(b) consiste en construir los circuitos de selección con puertas de transmisión. Esta implementación, mostrada para un selector de 4 a 1 en la Figura 4-18, usa puertas de transmisión como conmutadores. El circuito con puertas de transmisión proporciona un camino de transmisión entre cada entrada *I* y la salida *Y* cuando las dos entradas de selección de las puertas de transmisión del camino tienen el valor 1 en el terminal no negado y un 0 en el terminal negado. Si los valores se cambian en la entrada de selección, una de las entradas de transmisión del camino se convierte en un circuito abierto y el camino desaparece. Las dos entradas de selección S_1 y S_0 controlan los caminos de transmisión en el circuito con puertas de transmisión. Por ejemplo, si $S_0 = 0$ y $S_1 = 0$, existe un camino entre I_0 y la salida *Y*, y las otras tres entradas son desconectadas por el resto del circuito. El coste de una puerta de transmisión es equivalente al de las puertas de una entrada. Por tanto, el coste para este multiplexor basado en puertas de transmisión es de 8.



(a)



(b)

□ **FIGURA 4-17**
Circuitos de selección usando drivers tri-estado

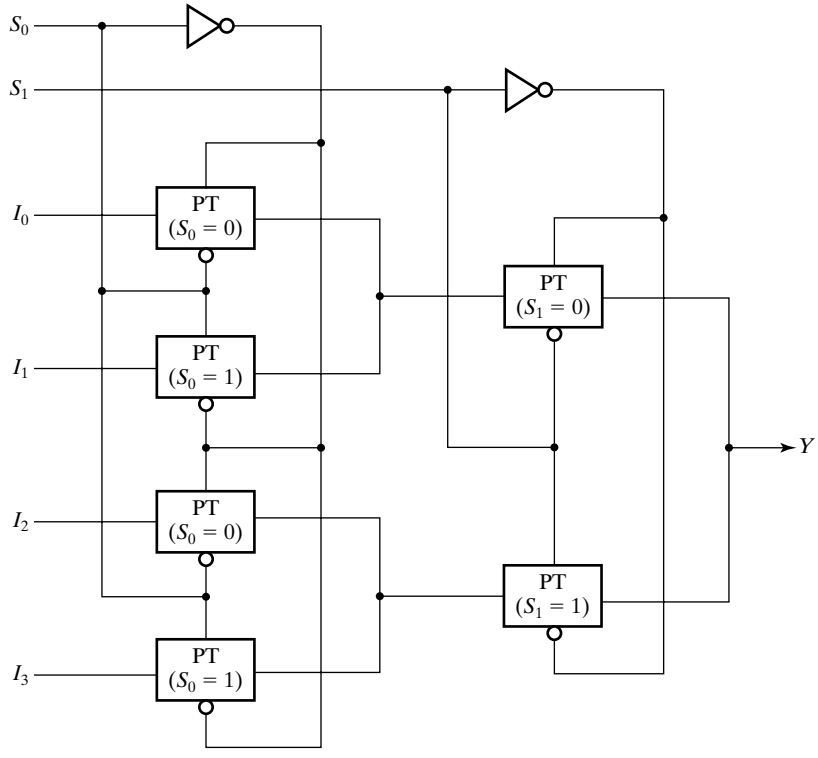
4-6 IMPLEMENTACIÓN DE FUNCIONES COMBINACIONALES

Los decodificadores y multiplexores pueden usarse para implementar funciones booleanas. Además puede considerarse que los dispositivos lógicos programables introducidos en el Capítulo 3 contienen bloques capaces de implementar funciones lógicas. En esta sección discutiremos el uso de decodificadores, multiplexores, memorias de sólo lectura (ROMs), arrays lógicos programables (PLAs), arrays de lógica programable (PALs), y tablas de búsqueda para implementar funciones lógicas combinacionales.

Empleando decodificadores

Un decodificador proporciona mini términos de n variables de entrada. Puesto que cualquier función booleana puede expresarse como una suma de mini términos, uno puede usar un decodificador para generar los mini términos y combinarlos con una puerta OR externa para implementar la función como una suma de mini términos. De esta manera, cualquier circuito combinacional con n entradas y m salidas puede implementarse con un decodificador n a 2^n y m puertas OR.

El procedimiento para implementar un circuito combinacional mediante un decodificador y puertas OR requiere expresar cada función booleana como una suma de mini términos. Esta



□ **FIGURA 4-18**
Multiplexor 4 a 1 usando puertas de transmisión

forma puede obtenerse a partir una tabla de verdad o construyendo el Mapa de Karnaugh de cada función. El decodificador se elige o diseña de modo que genere todos los mini términos de las variables de entrada. Las entradas de cada puerta OR se conectan a las salidas del decodificador correspondientes a los minitérminos de la función. Este proceso se muestra en el siguiente ejemplo.

EJEMPLO 4-6 Implementación de un sumador binario con decodificadores y puertas OR

En el Capítulo 1, vimos la suma binaria. La Tabla 4-8 da el valor del bit de suma *S* y el de acarreo *C* para cada posición en función de los bits *X* e *Y* de los operandos y el acarreo proveniente de la derecha *Z*.

De esta tabla de verdad se obtienen las funciones para este circuito combinacional, expresadas como suma de minitérminos:

$$S(X, Y, Z) = \Sigma m(1, 2, 4, 7)$$
$$C(X, Y, Z) = \Sigma m(3, 5, 6, 7)$$

Puesto que hay 3 entradas y un total de 8 mini términos, necesitaremos un decodificador 3 a 8. La implementación se muestra en la Figura 4-19. El decodificador genera los 8 mini términos de las entradas *X*, *Y* y *Z*. La puerta OR para la salida *S* obtiene la suma lógica de los mini términos 1, 2, 4 y 7. La puerta OR para la salida *C* obtiene la suma lógica de los minitérminos 3, 5, 6 y 7. El minitérmino 0 no se usa.

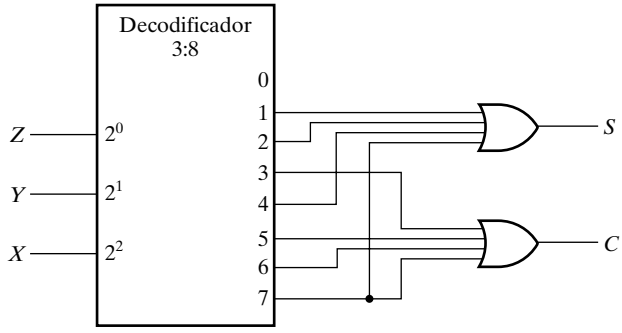
□ **TABLA 4-8**
Tabla de verdad de un sumador de 1 bit

X	*Y*	*Z*	*C*	*S*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

■

Una función con un mayor número de mini términos necesitará una puerta OR con mayor número de entradas. La negada de una función que tenga k mini términos puede ser expresada con $2^k - k$ minitérminos. Si el número de minitérminos en una función F es mayor que $2^k/2$, entonces la función negada $\overline{F}$ puede expresarse con un menor número de mini términos. En tal caso, resulta más ventajoso emplear puertas NOR en lugar de puertas OR. La parte OR de la puerta NOR genera la suma lógica de los mini términos de $\overline{F}$. La parte inversora de la puerta NOR obtiene F.

El método del decodificador puede emplearse para implementar cualquier circuito combinacional. Sin embargo, esta implementación debe ser comparada con otras posibles implementaciones para determinar la mejor solución posible. El método del decodificador puede proporcionar la mejor solución, especialmente si el circuito combinacional tiene algunas salidas dependientes de las mismas entradas y cada función de salida se expresa mediante un número pequeño de minitérminos.



□ **FIGURA 4-19**
Implementación de un sumador binario usando un decodificador

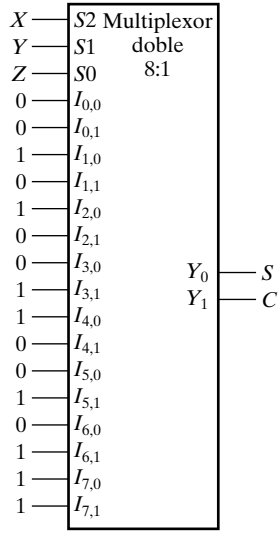
Empleando multiplexores

En la Sección 4-5 aprendimos cómo implementar un multiplexor empleando un decodificador y una puerta AND-OR $m \times 2$. El decodificador del multiplexor genera los minitérminos de las entradas de selección. La puerta AND-OR proporciona los circuitos de habilitación que determinan qué minitérminos son conectados a la puerta OR, empleando las entradas de información (I_i) como señales habilitadoras. Si la entrada I_i es 1, el minitérmino m_i es conectado a la

puerta OR, y si la entrada I_i es 0, el minitérmino m_i es sustituido por un 0. La asignación de valores aplicada a las entradas *I* proporciona un método de implementación de funciones booleanas de *n* variables empleando un multiplexor con *n* entradas de selección y 2^n entradas de datos, una por cada mini término. Además, una función con *m* salidas puede incrementarse empleando la asignación de valores en un multiplexor con vectores de información de *m* bits en lugar de un único bit individual, tal como ilustra el próximo ejemplo.

EJEMPLO 4-7 Implementación de un sumador binario con multiplexores

Los valores para las salidas *S* y *C* de un sumador binario de 1 bit se dan en la tabla de verdad de la Tabla 4-8 y se pueden generar empleando asignación de valores a las entradas de información de un multiplexor. Puesto que hay 3 entradas de selección y un total de ocho minitérminos, necesitamos un multiplexor doble de 8 a 1 líneas para implementar las dos salidas, *S* y *C*. La implementación se basa en la tabla de verdad mostrada en la Figura 4-20. Cada par de valores, como (0, 1) en ($I_{1,1}$, $I_{1,0}$), se toma directamente de la fila correspondiente de las últimas dos columnas de la tabla.

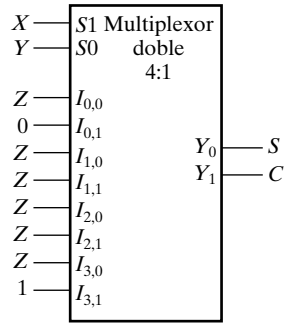


□ **FIGURA 4-20**
Implementación de un sumador binario de 1 bit con un doble multiplexor de 8 a 1 ■

Hay otro método más eficiente que implementa una función booleana de *n* variables con un multiplexor que tiene sólo $n - 1$ entradas de selección. Las primeras $n - 1$ variables de la función se conectan a las entradas de selección del multiplexor. Las restantes variables de la función se emplean como entradas de información. Si la última variable es *Z*, cada entrada de datos del multiplexor será *Z*, $\overline{Z}$, 1 o 0. La función puede implementarse conectando directamente las cuatro funciones básicas de la Tabla 4-1 a las entradas de datos del multiplexor. El próximo ejemplo muestra este procedimiento.

EJEMPLO 4-8 Implementación alternativa con multiplexores de un sumador binario

Esta función puede implementarse con un multiplexor doble de 4 a 1, como muestra la Figura 4-21. El proceso de diseño puede ilustrarse considerando la suma *S*. Las dos variables *X* e *Y*



□ **FIGURA 4-21**
Implementación de un sumador binario de 1 bit con un multiplexor doble de 8 a 1

se aplican como líneas de selección en el siguiente orden: X se conecta a la entrada S_1, e Y se conecta a la entrada S_0. A partir de la tabla de verdad se determinan los valores para las líneas de entrada de la función. Cuando $(X, Y) = 00$, la salida S es igual a Z porque $S = 0$ cuando $Z = 0$ y $S = 1$ cuando $Z = 1$. Esto requiere que la variable Z se aplique como información de entrada I_{00}. El funcionamiento del multiplexor es tal que, cuando $(X, Y) = 00$, la entrada de información I_{00} tiene un camino hacia la salida que hace que S sea igual a Z. De manera similar podemos determinar las entradas necesarias para las líneas I_{10}, I_{20} e I_{30} a partir de los valores de S cuando (X, Y) sea 01, 10 y 11, respectivamente. Puede usarse una aproximación similar para determinar los valores para I_{01}, I_{11}, I_{21} e I_{31}. ■

El procedimiento general para implementar cualquier función booleana de n variables con un multiplexor con $n - 1$ entradas de selección y 2^{n-1} entradas de datos se expone en el siguiente ejemplo. Primero se pasa la función booleana a una tabla de verdad. Las $n - 1$ primeras variables de la tabla se aplican como entradas de selección al multiplexor. Para cada combinación de estas variables de selección, evaluamos la salida como una función de la última variable. Esta función puede ser 0, 1, la variable, o el complemento de la variable. Estos valores se aplican a la entrada de datos apropiada. Este proceso se ilustra mediante el próximo ejemplo.

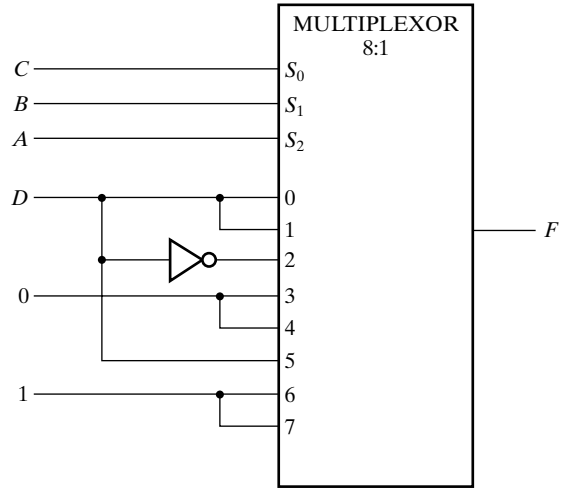
EJEMPLO 4-9 Implementación con multiplexores de una función de 4 variables

Como segundo ejemplo, considere la implementación de la siguiente función Booleana:

$$F(A, B, C, D) = \Sigma m(1, 3, 4, 11, 12, 13, 14, 15)$$

Esta función se implementa con un multiplexor 8 × 1, como muestra la Figura 4-22. Para obtener el resultado correcto, las variables de la tabla de verdad se conectan a las entradas de selección S_2, S_1 y S_0 en el orden en que aparecen en la tabla (es decir, tal que A se conecta a S_2, B se conecta a S_1 y C se conecta a S_0). Los valores de las entradas de datos vienen determinados por la tabla de verdad. El número en la línea de información se determina a partir de las combinaciones binarias de A, B y C. Por ejemplo, cuando $(A, B, C) = 101$, la tabla de verdad muestra que $F = D$ por lo que la variable D se aplica a la entrada de información I_5. Las constantes binarias 0 y 1 corresponden a dos valores fijos de señal. Recuerde que, de la Sección 4-2, en los esquemas lógicos estos valores constantes se reemplazaban por los símbolos de masa y alimentación que aparecen en la Figura 4-2.

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = \overline{D}$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	



□ **FIGURA 4-22**
Implementación de una función de 4 entradas con un multiplexor ■

Empleando memorias de sólo lectura

En base a los principios sobre decodificadores y multiplexores tratados hasta ahora hay dos aproximaciones para implementar memorias de solo lectura. Una aproximación se basa en el empleo de un decodificador y puertas OR. Insertando puertas OR en paralelo, una por cada salida de la ROM, para sumar los mini términos de las funciones booleanas, somos capaces de genera cualquier circuito combinacional que deseemos. Las ROM pueden ser vistas como dispositivos que incluyen un decodificador y puertas OR dentro de una única unidad. Cerrando las conexiones de las entradas de una puerta OR para los mini términos de la función, las salidas de la ROM pueden programarse para representar las funciones booleanas de las variables de salida de un circuito combinacional. Una solución alternativa está basada en la asignación de valores fijos a un multiplexor de múltiples bits. Los valores I_i se emplean como señales de habilitación que determinan qué mini términos están conectados a las puertas OR del multiplexor. Esto se ilustra en el Ejemplo 4-8 que equivale a una ROM de 3 entradas y 2 salidas. La «programación» de este enfoque ROM se realiza aplicando la tabla de verdad a las entradas de información del multiplexor. Dado que el enfoque basado en un decodificador y puertas OR es simplemente un modelo diferente, éste también, puede «programarse» usando la tabla de verdad para determinar las conexiones entre el decodificador y las puertas OR. De esta manera, en la práctica, cuando un circuito combinacional es diseñado por medio de una ROM, no se diseña necesariamente la lógica ni se muestran las conexiones internas dentro de la unidad. Todo lo que el diseñador tiene que hacer es especificar la ROM particular por su nombre y proporcionar su tabla de verdad. La tabla de verdad da toda la información necesaria para programar la ROM. No es necesario que acompañe a la tabla ningún diagrama lógico interno. El Ejemplo 4-10 muestra este uso para una ROM.

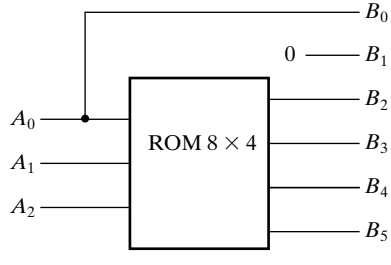
EJEMPLO 4-10 Implementando un circuito combinacional con una ROM

Diseñe un circuito combinacional usando una ROM. El circuito acepta números de 3 bits y genera un número de salida binario igual al cuadrado del número de entrada.

El primer paso del diseño es obtener la tabla de verdad del circuito combinacional. En la mayoría de los casos realizaremos una tabla de verdad parcial para la ROM, empleando determinadas propiedades en el cálculo de las salidas de las variables. La Tabla 4-9 es la tabla de verdad para el circuito combinacional. Se necesitan tres entradas y seis salidas para acomodar todos los posibles números binarios. Notemos como la salida B_0 es siempre igual a la entrada A_0, por lo que no hay que generar B_0 con la ROM. Es más, la salida B_1 es siempre 0, de modo que esta salida es una constante conocida. Entonces sólo es necesario generar cuatro salidas con la ROM; las otras dos salidas ya están obtenidas. El tamaño mínimo para la ROM ha de ser de 3 entradas y cuatro salidas. Las tres entradas especifican ocho palabras; por lo que la ROM ha de tener un tamaño de 8 × 4. La Figura 4-23 muestra la implementación de la ROM. Las tres entradas especifican ocho palabras de cuatro bits cada una. El diagrama de bloques de la Figura 4-23(a) muestra las conexiones necesarias del circuito combinacional. La tabla de verdad de la Figura 4-23(b) especifica la información necesaria para programar la ROM.

□ **TABLA 4-9**
Tabla de verdad para el circuito del Ejemplo 4-10

Entradas			Salidas						Decimal
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49



(a) Diagrama de bloques

A_2	A_1	A_0	B_5	B_4	B_3	B_2
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

(b) Tabla de verdad de la ROM

□ **FIGURA 4-23**
Implementación con ROM del Ejemplo 4-10

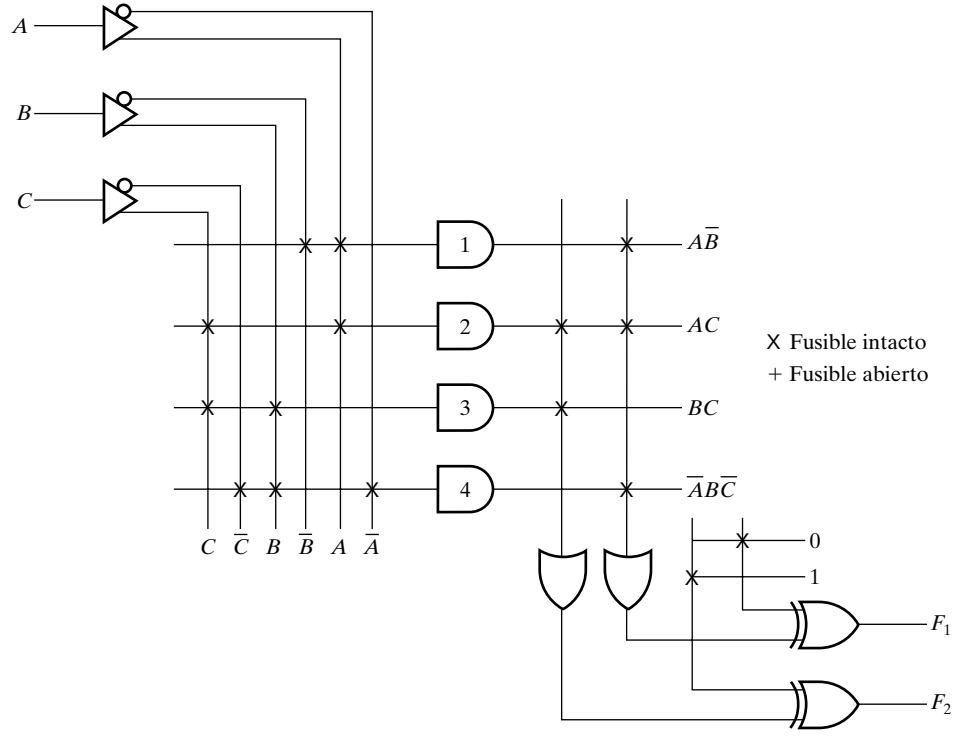
■

Los dispositivos ROM son muy empleados para implementar circuitos combinacionales complejos directamente a partir de sus tablas de verdad. Son muy útiles para convertir desde un código, por ejemplo el código Gray, a otro código, como el BCD. Pueden generar operaciones aritméticas complejas, tales como la multiplicación o la división, y en general, se usan en aplicaciones que requieran un moderado número de entradas y un gran número de salidas.

Usando arrays lógicos programables

El PLA es un concepto muy similar al de la ROM, excepto que el PLA no proporciona la decodificación de todas las variables, por tanto no genera todos los mini términos. El decodificador es sustituido por un array de puertas AND, cada una de las cuales puede ser programada para generar cualquier producto de términos de las variables de entrada. Los términos producto se conectan selectivamente a puertas OR, como en una ROM, para obtener las sumas de productos de las funciones requeridas.

Este mapa de fusibles de un PLA puede especificarse en forma tabular. Por ejemplo, la tabla de programación que determina el funcionamiento del PLA de la Figura 4-24 se enumera en la Tabla 4-10. La Tabla tiene tres secciones. La primera sección enumera el número de los términos producto. La segunda sección especifica los caminos necesarios entre las entradas y las puertas AND. La tercera sección especifica los caminos entre las puertas AND y las puertas OR. Para cada variable de salida, se indica con T (true-verdadero) o C (complemento), si la salida debe ser negada medante una puerta XOR. Los términos producto enumerados a la izquierda



□ **FIGURA 4-24**
PLA con 3 entradas, 4 productos y 2 salidas

□ **TABLA 4-10**
Tabla de programación del PLA de la Figura 4-24

		Entradas			Salidas	
	Término producto	***A***	***B***	***C***	**(T) F_1**	**(C) F_2**
$A\bar{B}$	1	1	0	—	1	—
AC	2	1	—	1	1	1
BC	3	—	1	1	—	1
$\bar{A}B\bar{C}$	4	0	1	0	1	—

no forman parte de la tabla; solo se ha incluido como referencia. Para cada término producto las entradas se marcan con 1, 0 o —. Si una variable aparece en el producto de términos en su forma no negada, su correspondiente variable de entrada se marca con 1. Si la variable en el producto de términos aparece complementada, la variable de entrada correspondiente estará marcada como 0. Si la variable está ausente en el producto de términos, se marcará con —.

Los caminos entre las entradas y las puertas AND se especifican en la columna encabezada como *entradas* de la tabla. Un 1 en la columna de entrada indica un circuito CERRADO desde la variable de entrada hasta la puerta AND. Un 0 en la columna de entrada indica un circuito CERRADO desde el complemento de la variable de entrada hasta la puerta AND. Un — indica un circuito ABIERTO tanto para la variable de entrada como para su complementaria. Se supone que un terminal ABIERTO desde una entrada hasta una puerta AND se comporta como un 1. Los caminos entre las puertas AND y OR se especifican bajo la columna encabezada como *salidas*. Las variables de salida se marcan con 1 para aquellos términos producto que están incluidos en la función. Cada término producto que presenta un 1 en su columna de salida necesita un camino CERRADO desde la salida de la puerta AND hasta la entrada de la puerta OR. Aquellos términos producto marcados con un —, indican un circuito ABIERTO. Se supone que un terminal abierto en la entrada de una puerta OR se comporta como un 0. Finalmente, una salida T dicta que la otra entrada de la correspondiente puertas XOR se conecta a 0, y una C especifica una conexión a 1.

El tamaño del PLA se especifica mediante el número de entradas, el número de términos producto y el número de salidas. Un PLA típico tiene 16 entradas, 48 términos producto y 8 salidas. Para n entradas, k producto y m salidas, la lógica interna del PLA consiste en n buffers-inversores, k puertas AND, m puertas OR y m puertas XOR. Existen $2n \times k$ posibles conexiones programables entre las entradas y el array de AND, $k \times n$ conexiones programables entre los arrays AND y OR, y m conexiones programables asociadas a las puertas XOR.

En el diseño de un sistema digital mediante PAL no es necesario mostrar las conexiones internas de la unidad, tal y como se ha hecho en la Figura 4-24. Lo único que se necesita es la tabla de programación con la que programar dicho PLA para obtener la lógica requerida. Como con una ROM, la PLA puede ser programable por máscara o en campo.

En la implementación de un circuito combinacional con PLA, hay que prestar especial atención a la obtención del menor número de productos distintos, dado que de este modo puede reducirse la complejidad del circuito. Se pueden disminuir los productos mediante simplificación de la función booleana hasta lograr un mínimo número de términos. Puesto que en un PLA se dispone en todo momento de todas las variables de entrada, el número de literales en cada término es menos importante. Sin embargo, es deseable para evitar literales extra que pueden

causar problemas durante el test del circuito y que pueden reducir su velocidad. Tanto la forma directa como la complementaria de cada función deben simplificarse para ver cuál de ellas se puede expresar con menos productos y cuáles contienen productos comunes a otras funciones. Este proceso se muestra en el Ejemplo 4-11.

EJEMPLO 4-11 Implementando un circuito combinacional usando un PLA

Implemente las dos funciones booleanas siguientes con PLA:

$$F_1(A, B, C) = \Sigma m(0, 1, 2, 4)$$

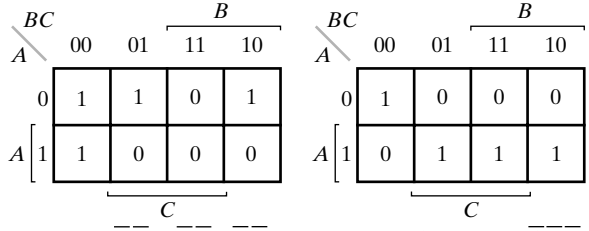
$$F_2(A, B, C) = \Sigma m(0, 5, 6, 7)$$

Las dos funciones se simplifican con ayuda de los mapas de la Figura 4-25. Tanto las funciones directas como sus complementarias se simplifican en forma de sumas de productos. Las combinaciones que dan el menor número de productos son:

$$F_1 = \overline{AB + AC + BC}$$

$$F_2 = AB + AC + \bar{A}\bar{B}\bar{C}$$

La simplificación proporciona cuatro productos distintos: AB, AC, BC y $\bar{A}\bar{B}\bar{C}$. La tabla de programación del PLA para esta combinación es la mostrada en la figura. Observe que la salida F_1 es la salida directa y se designa con una C en la tabla. Esto es así porque $\bar{F}_1$ se genera con el circuito AND-OR y está disponible a la salida de la puerta OR. La puerta XOR complementa la función $\bar{F}_1$ para producir la salida F_1.



$F_1 = \bar{A}\bar{B} + \bar{A}\bar{C} + \bar{B}\bar{C}$
$\bar{F}_1 = AB + AC + BC$

$F_2 = AB + AC + \bar{A}\bar{B}\bar{C}$
$\bar{F}_2 = \bar{A}\bar{C} + \bar{A}B + A\bar{B}\bar{C}$

Tabla de programación del PLA

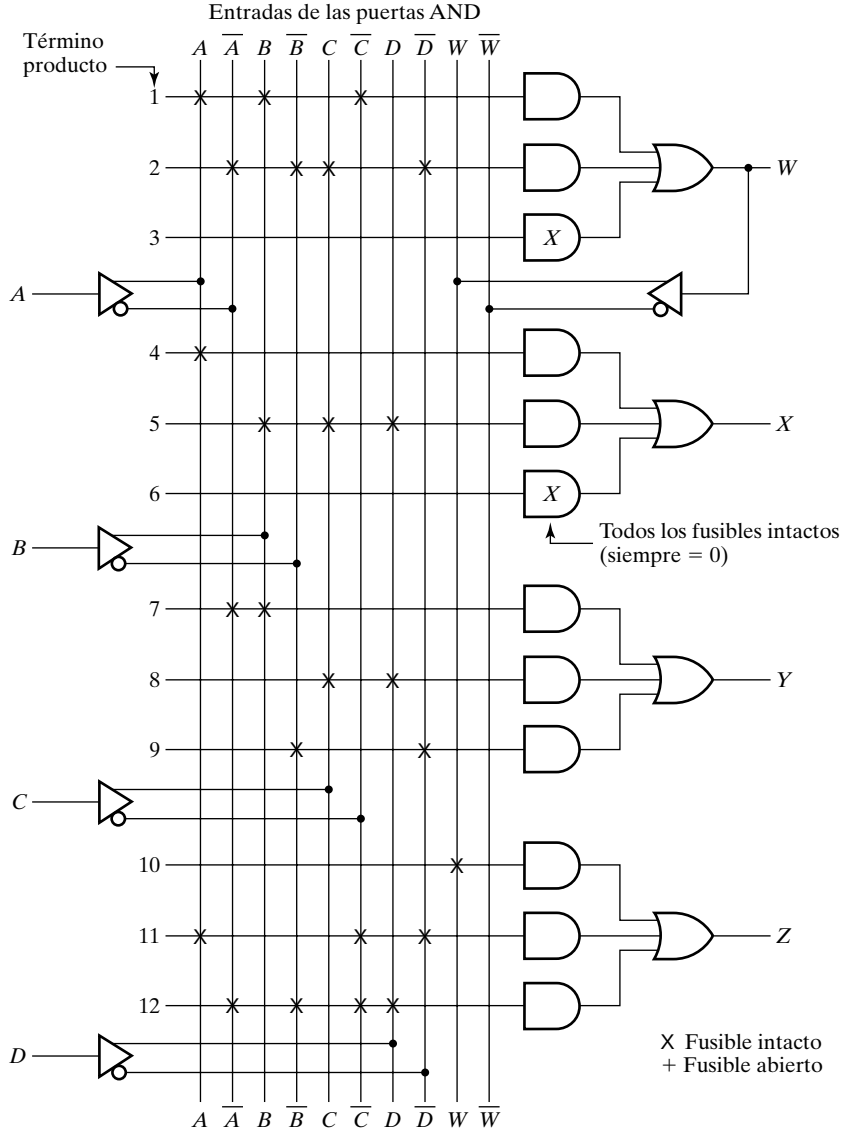
		Entradas			Salidas (C)	(T)
	Término producto	A	B	C	F_1	F_2
AB	1	1	1	–	1	1
AC	2	1	–	1	1	1
BC	3	–	1	1	1	–
$\bar{A}\,\bar{B}\,\bar{C}$	4	0	0	0	–	1

□ **FIGURA 4-25**
Solución al Ejemplo 4-11

■

Usando arrays de lógica programable

Al diseñar con un dispositivo PAL, las funciones booleanas deben simplificarse para encajar en cada sección, como se ilustra en el PAL de ejemplo de la Figura 4-26. Al contrario de como ocurría con el PLA, un producto no puede compartirse entre dos o más puertas OR. Por ello cada función debe simplificarse por sí misma, sin tener en cuenta los productos comunes. El número de productos en cada sección es fijo, y si el número de términos de la función es demasiado grande, puede ser necesario usar dos o más secciones para implementar la función booleana. En tal caso, los términos comunes pueden ser útiles. Este proceso se ilustra en el Ejemplo 4-12.



□ **FIGURA 4-26**
Mapa de conexiones de una PAL® para el Ejemplo 4-12

EJEMPLO 4-12 Implementación de un circuito combinacional usando un PAL

Como un ejemplo de empleo de un dispositivo PAL para el diseño de un circuito combinacional, considere las funciones booleanas siguientes dadas en forma de suma de mini términos:

$$W(A, B, C, D) = \Sigma m(2, 12, 13)$$
$$X(A, B, C, D) = \Sigma m(7, 8, 9, 10, 11, 12, 13, 14, 15)$$
$$Y(A, B, C, D) = \Sigma m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$$
$$Z(A, B, C, D) = \Sigma m(1, 2, 8, 12, 13)$$

Simplificando las cuatro funciones a un número mínimo de términos resultan las siguientes funciones booleanas:

$$W = AB\bar{C} + \bar{A}\bar{B}CD$$
$$X = A + BCD$$
$$Y = \bar{A}B + CD + \bar{B}\bar{D}$$
$$Z = AB\bar{C} + \bar{A}\bar{B}CD + A\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D$$
$$= W + A\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D$$

Observe que la función *Z* tiene cuatro productos. La suma lógica de dos de estos términos es igual a *W*. Entonces, usando *W*, es posible reducir el número de términos para *Z* de cuatro a tres, para que las funciones puedan encajar en el dispositivo PAL de la Figura 4-26.

La tabla de programación de la PAL es similar a la tabla usada para el PLA, excepto que sólo necesitan ser programadas las entradas de las puertas AND. La Tabla 4-11 enumera la tabla de programación para la PAL para las cuatro funciones booleanas anteriores. La tabla se divide en cuatro secciones con tres productos cada una, de acuerdo con el dispositivo PAL de la Figura 4-26. Las primeras dos secciones sólo necesitan dos productos de términos para llevar a cabo la función booleana. Poniendo *W* en la primera sección del dispositivo, la conexión de realimentación desde F1 hasta el array permite reducir la función *Z* a tres términos.

El mapa de conexión para el dispositivo PAL, como se ha especificado en la tabla de programación, se muestra en la Figura 4-26. Por cada 1 o 0 de la tabla, marcamos la intersección correspondiente en el diagrama con el símbolo de una conexión cerrada. Para cada —, marcamos como abiertas tanto las entradas directas como las complementarias. Si la puerta AND no se usa, dejamos todas sus entradas como circuitos cerrados. Puesto que la entrada correspondiente recibe una señal y su complemento, tenemos el $A\bar{A} = 0$, y la salida de la puerta AND siempre es 0. ■

Empleando tablas de búsqueda

Los FPGAs y los dispositivos lógicos programables complejos (CPLDs), a menudo usan tablas de búsqueda *(LookUp Tables-LUTs)* para implementar su lógica. Programar una única función de *m* entradas es igual que programar un ROM de una sóla salida (es decir, la tabla de busqueda simplemente almacena la tabla de verdad de la función). Una tabla de búsqueda de *m* entradas puede implementar cualquier función de *m* o menos variables. Típicamente, $m = 4$. La clave del problema de la programación de las tablas de búsqueda está en tratar con funciones de más de *m* variables de entrada. También es importante la compartición de las tablas de búsqueda entre múltiples funciones. Estos problemas pueden tratarse empleando transformaciones lógicas de

□ **TABLA 4-11**
Tabla de programación del PAL® del Ejemplo 4.12

Producto terminado	Entradas de las puertas AND					Salidas
	A	***B***	***C***	***D***	***W***	
1	1	1	0	—	—	$W = AB\bar{C}$
2	0	0	1	0	—	$+ \bar{A}\bar{B}C\bar{D}$
3	—	—	—	—	—	
4	1	—	—	—	—	$X = A$
5	—	1	1	1	—	$+ BCD$
6	—	—	—	—	—	
7	0	1	—	—	—	$Y = \bar{A}B$
8	—	—	1	1	—	$+ CD$
9	—	0	—	0	—	$+ B\bar{D}$
10	—	—	—	—	1	$Z = W$
11	1	—	0	0	—	$+ A\bar{C}\bar{D}$
12	0	0	0	1	—	$+ \bar{A}\bar{B}\bar{C}D$

varios niveles, principalmente la descomposición y la extracción. La meta de la optimización es implementar la función o funciones usando el menor número posible de LUT, con la limitación de que cada LUT puede implementar funciones de, a lo sumo, *m* variables. Esto puede lograrse encontrando un menor número posible de ecuaciones, cada una de *m* variables como máximo, que implementan la función o funciones deseadas. Este proceso se ilustra para funciones de una única salida y de múltiples salidas con $m = 4$ en los siguiente ejemplos.

EJEMPLO 4-13 Implementación con tablas de búsqueda de una función de una única salida

Implemente la siguiente función booleana empleando tablas de búsqueda:

$$F_1(A, B, C, D, E, F, G, H, I) = ABCDE + \bar{F}GHI\bar{D}\bar{E}$$

El número de variables de entrada para una función se llama *soporte*, *s*, de la función. El soporte para F_1 es $s = 9$. Aparentemente el número mínimo, *k*, de tablas de búsqueda necesario es por lo menos 9/4 (es decir, $k = 3$). Además, para una función de *m*-salidas, el número mínimo de tablas de búsqueda para una función de una única salida debe obedecer la relación más estricta $mk \geqslant s + k - 1$, de modo que *k* debe satisfacer $4k \geqslant 9 + k - 1$. Resolviendo, $k = 3$, de modo que buscaremos una descomposición de F_1 en tres ecuaciones, cada una con, como máximo, $s = 4$. Factorizando F_1, obtenemos

$$F_1 = (ABC)DE + (\bar{F}GHI)\bar{D}\bar{E}$$

Basándonos en esta ecuación, F_1 puede descomponerse en tres ecuaciones con $s \leqslant 4$:

$$F_1(D, E, X_1, X_2) = X_1DE + X_2\bar{D}\bar{E}$$
$$X_1(A, B, C) = ABC$$
$$X_2(F, G, H, I) = \bar{F}GHI$$

Cada una de estas tres ecuaciones puede implementarse mediante una LUT, dando una implementación óptima para F_1. ■

EJEMPLO 4-14 Implementación de una función de varias salidas con tablas de búsqueda

Implemente el siguiente par de funciones booleanas con tablas de búsqueda:

$$F_1(A, B, C, D, E, F, G, H, I) = ABCDE + \bar{F}GHI\bar{D}\bar{E}$$
$$F_2(A, B, C, D, E, F, G, H, I) = ABCEF + \bar{F}GHI$$

Cada una de estas funciones requiere un soporte $s = 9$. Por tanto se necesitan por lo menos tres LUTs para cada función. Pero dos de las LUTs pueden compartirse, de modo que el número mínimo de LUTs necesarias es $k = 6 - 2 = 4$. Factorizando F_2 para poder compartir ecuaciones con la descomposición de F_1 del ejemplo anterior se obtiene:

$$F_2 = (ABC)EF + (\bar{F}GHI)$$

Esto produce una descomposición para F_1 y F_2:

$$F_1(D, E, X_1, X_2) = X_1DE + X_2\bar{D}\bar{E}$$
$$F_2(E, F, X_1, X_2) = X_1EF + X_2$$
$$X_1(A, B, C) = ABC$$
$$X_2(F, G, H, I) = \bar{F}GHI$$

En este caso, la extracción requiere cuatro LUTs, el número mínimo. Por regla general no puede garantizarse la localización de una descomposición o extracción que requiera el mínimo número calculado de LUTs. ■

4-7 HDL REPRESENTACIÓN PARA CIRCUITOS COMBINACIONALES—VHDL

Dado que un HDL se emplea para describir y diseñar hardware, es muy importante tener presente cómo se escribe en ese lenguaje el hardware involucrado. Esto es particularmente crítico si la descripción HDL debe ser sintetizada. Por ejemplo, si se ignora el hardware que se generará, es muy fácil especificar una estructura de puertas enorme y compleja usando × (multiplicación) cuando todo lo que se hubiera necesitado es una estructura mucho más simple con tan sólo unas pocas puertas. Por esta razón, inicialmente hacemos hincapié en la descripción detallada del hardware con VHDL, y procedemos después a descripciones más abstractas de niveles superiores.

Los ejemplos seleccionados en este capítulo son útiles para introducir VHDL como un medio alternativo para representar detalladamente circuitos digitales. Inicialmente mostramos las descripciones estructurales VHDL que sustituyen al esquemático para el decodificador 2 a 4 con

habilitación de la Figura 4-10. Este ejemplo y otro que usa el multiplexor 4 a 1 de la Figura 4-14, ilustran muchos de los conceptos fundamentales del VHDL. Después presentaremos descripciones VHDL de nivel funcional superior y descripciones VHDL de comportamiento para estos circuitos que ilustrarán más conceptos fundamentales de VHDL.

EJEMPLO 4-15 VHDL estructural para un decodificador 2 a 4

La Figura 4-27 muestra una descripción VHDL para el circuito decodificador 2 a 4 de la Figura 4-10, en la página 143. Este ejemplo se usará para mostrar varias características generales del VHDL así como la descripción estructural de circuitos.

El texto entre dos - - y el final de línea se interpreta como un comentario. Así que la descripción en la Figura 4-27 empieza con dos líneas de comentarios que identifican la descripción y su relación con la Figura 4-10. Para ayudar en la discusión de esta descripción se han agregado comentarios, a la derecha, que proporcionan los números de línea. Como lenguaje que es, VHDL tiene una sintaxis que describe de forma precisa las estructuras válidas que pueden usarse. Este ejemplo ilustrará muchos aspectos de la sintaxis. En particular véase el uso de punto y coma, comas y dos puntos en la descripción.

```
-- Decodificador 2-a-4 con habilitación: descripción VHDL estructural        --  1
-- (véase la Figura 4-10 para el diagrama lógico)                            --  2
library ieee, lcdf_vhdl;                                                     --  3
 se ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;                       --  4
entity decoder_2_to_4_w_enable is                                            --  5
  port (EN, A0, A1: in std_logic;                                            --  6
        D0, D1, D2, D3: out std_logic);                                      --  7
end decoder_2_to_4_w_enable;                                                 --  8
                                                                             --  9
architecture structural_1 of decoder_2_to_4_w_enable is                      -- 10
  component NOT1                                                             -- 11
    port(in1: in std_logic;                                                  -- 12
         out1: out std_logic);                                               -- 13
  end component;                                                             -- 14
  component AND2                                                             -- 15
    port(in1, in2: in std_logic;                                             -- 16
         out1: out std_logic);                                               -- 17
  end component;                                                             -- 18
  signal A0_n, A1_n, N0, N1, N2, N3: std_logic;                              -- 19
  begin                                                                      -- 20
    g0: NOT1 port map (in1 => A0, out1 => A0_n);                             -- 21
    g1: NOT1 port map (in1 => A1, out1 => A1_n);                             -- 22
    g2: AND2 port map (in1 => A0_n, in2 => A1_n, out1 => N0);                -- 23
    g3: AND2 port map (in1 => A0, in2 => A1_n, out1 => N1);                  -- 24
    g4: AND2 port map (in1 => A0_n, in2 => A1,out1 => N2);                   -- 25
    g5: AND2 port map (in1 => A0, in2 => A1, out1 => N3);                    -- 26
    g6: AND2 port map (in1 => EN, in2 => N0, out1 => D0);                    -- 27
    g7: AND2 port map (in1 => EN, in2 => N1, out1 => D1);                    -- 28
    g8: AND2 port map (in1 => EN, in2 => N2, out1 => D2);                    -- 29
    g9: AND2 port map (in1 => EN, in2 => N3, out1 => D3);                    -- 30
end structural_1;                                                            -- 31
```

□ **FIGURA 4-27**
Descripción estructural en VHDL de un decodificador 2 a 4

Inicialmente, saltamos las líneas 3 y 4 de la descripción para centrarnos en la estructura global. La línea 5 comienza con la declaración de una *entidad* que es la unidad fundamental de un diseño VHDL. En VHDL por cada uno de los símbolos del esquemático necesitamos darle un nombre al diseño y definir sus entradas y salidas. Esta es la función de la *declaración de entidad*. En VHDL, **entity** e **is** son palabras claves. Las palabras clave que nosotros mostramos en tipo negrita tienen un significado especial y no pueden usarse para nombrar objetos tales como las entidades, entradas, salidas o señales. La sentencia **entity** `decoder_2_to_4_w_enable` **is** declara que existe un diseño con el nombre `decoder_2_to_4_w_enable`. VHDL no es sensible a las mayúsculas (es decir, los nombres y palabras claves no se distinguen por estar escritas en letras mayúsculas o minúsculas). `DECODER_2_4_W_ENABLE` es lo mismo que `Decoder_2_4_w_Enable` y `decoder_2_4_w_enable`.

Lo siguiente, una *declaración de puertos* en las líneas 6 y 7 se usa para definir las entradas y salidas tal y como se haría en un símbolo de un esquemático. Para el diseño del ejemplo, hay tres señales de entrada: `EN`, `A0` y `A1`. El modo **in** denota que estas son entradas. Igualmente, se designan `D0`, `D1`, `D2` y `D3` como salidas mediante el modo **out**. VHDL es un lenguaje fuertemente-tipado, de modo que debe declararse el tipo de todas las entradas y salidas. En este caso, el tipo es `std_logic` que representa *lógica estándar*. Esta declaración del tipo especifica los valores que pueden aparecer en las entradas y en las salidas, así como las operaciones que pueden aplicarse a las señales. La lógica estándar, entre sus nueve valores, incluye los valores binarios usuales 0 y 1 y dos valores adicionales X y U. X representa un valor desconocido, U un valor sin inicializar. Hemos escogido usar lógica normal, que incluye estos valores, por ser dichos valores empleados por las herramientas típicas de la simulación.

Para usar el tipo `std_logic`, es necesario definir los valores y las operaciones. Por conveniencia, se emplea un *package* (paquete) consistente en código VHDL precompilado. Normalmente, los packages se guardan en un directorio llamado *library* que es compartido por algunos, o por todos, los usuarios de la herramienta. Para el `std_logic`, el paquete básico es `ieee.std_logic_1164`. Este paquete define los valores y las operaciones lógicas básicas para los tipos `std_ulogic` y `std_logic`. Para usar `std_logic`, incluimos en la línea 3 una llamada a la **library** (librería) de paquetes llamada `ieee` e incluimos en la línea 4 conteniendo `ieee.std_logic_1164.`**all** para indicar que queremos usar todo (**all**) el paquete `std_logic_1164` de la librería `ieee`. También queremos usar todo el paquete `func_prims` de una librería adicional, `lcdf_vhdl`, que contiene descripciones VHDL de puertas lógicas básicas, biestables y latches. La librería `lcdf_vhdl` está disponible en ASCII y puede copiarse para este texto desde la página web del libro. Observe que las declaraciones de las líneas 3 y 4 están vinculadas con la entidad siguiente. Si se incluyera otra entidad que usase el tipo `std_logic` y los elementos del paquete `func_prims`, estas declaraciones deberían repetirse con anterioridad a la declaración de dicha entidad.

La declaración de entidad acaba con la palabra clave **end** seguida por el nombre de la entidad. Hasta aquí, hemos discutido el equivalente en VHDL para un símbolo de un circuito.

DESCRIPCIÓN ESTRUCTURAL Ahora, queremos especificar la función del circuito. Una representación particular de la función de una entidad se llama *arquitectura* (*architecture*) de la entidad. Así, los contenidos de la línea 10 declaran una arquitectura VHDL llamada `structural_1` para la entidad `decoder_2_to_4_w_enable`. A continuación aparecen los detalles de la arquitectura. En este caso usamos una *descripción estructural* que es equivalente al esquemático para el circuito dado en la Figura 4-10.

Primero, desde la línea 11 hasta la 18, declaramos los tipos de puertas que se van a usar como componentes de la descripción. Puesto que estamos construyendo esta arquitectura a par-

tir de puertas, declaramos un inversor llamado `NOT1` y una puerta AND de 2-entrada llamada `AND2` como *componentes*. Estos tipos de puertas son descripciones VHDL dentro del paquete `func_prims` que contiene la entidad y la arquitectura para cada una de las puertas. El nombre y la declaración de puertos de un componente deben ser idénticos a aquéllos de su entidad. Para `NOT1`, `port` indica que `in1` es el nombre de la entrada y `out1` es el nombre de la salida. La declaración del componente para `AND2` nombra `in1` e `in2` como las entradas y `out1` como la salida.

Lo siguiente que necesitamos, antes de especificar la interconexión de las puertas, que es equivalente a un netlist del circuito, es nombrar todos los nodos del circuito. Las entradas y salidas ya tienen nombres. Los nodos interiores son las salidas de los dos inversores y las cuatro puertas AND de más a la izquierda de la Figura 4-10. Estos nodos de salida se declaran como señales (*signals*) del tipo `std_logic`. `A0_n` y `A1_n` son las señales para las salidas de los dos inversores y `N0`, `N1`, `N2` y `N3` son las señales para las salidas de las cuatro puertas AND. Igualmente, todas las entradas y salidas declaradas como puertos son señales. En VHDL, hay señales y variables. Las variables se evalúan instantáneamente. Por contra, las señales se evalúan en algún instante futuro de tiempo. Este tiempo puede ser tiempo físico, como 2 ns después de ahora, o puede ser lo que se llama *tiempo delta* (*delta time*), en que una señal se evalúa un tiempo delta después del tiempo actual. El tiempo delta es equivalente a una cantidad infinitesimal de tiempo. Para el funcionamiento interno de los simuladores digitales típicos es esencial considerar algún retardo en la evaluación de las señales y, por supuesto, este tiempo, basado en el retardo de las puertas, modela de forma realista el comportamiento de los circuitos. Por simplicidad, típicamente simularemos los circuitos para comprobar si su diseño es correcto, no para detectar problemas en las prestaciones o retardos. Para estas simulaciones funcionales es más fácil igualar estos retardos a un tiempo delta. Así, nuestras descripciones circuitales VHDL no explicitarán ningún retardo, aunque pueden aparecer retardos en los bancos de prueba.

Tras la declaración de las señales internas, el grueso de la arquitectura empieza con la palabra clave **`begin`**. El circuito descrito consiste en dos inversores y ocho puertas AND de 2 entradas. En la línea 21 se coloca la etiqueta `g0` al primer inversor e indica que el inversor es el componente `NOT1`. Luego aparece un **`port map`** (mapeado de puertos) que asigna la entrada y salida del inversor a las señales a las que se conectan. Esta forma particular de *port map* usa el símbolo `=>` con el puerto de la puerta a la izquierda y la señal a la que se conecta a la derecha. Por ejemplo, la entrada `g0` del inversor es `A0` y la salida es `A0_n`. Desde la línea 22 hasta la línea 30 se etiquetan las nueve puertas restantes y se asignan las señales conectadas a sus entradas y salidas. Por ejemplo, en la línea 24, `A0` y `A1_n` son las entradas y `N1` es la salida. La arquitectura se completa con la palabra clave **`end`** seguida por su nombre `structural_1`. ■

EJEMPLO 4-16 VHDL estructural para un multiplexor 4 a 1

En Figura 4-28, la descripción estructural del multiplexor de 4 a 1 de la Figura 4-14 ilustra dos conceptos adicionales del VHDL: *std_logic_vector* y una aproximación alternativa al mapeado de los puertos.

En las líneas 6 y 7, en lugar de especificar `S` e `I` como entradas individuales del tipo *std_logic*, se especifican como vectores del tipo *std_logic_vector*. Para especificar vectores se usa un índice. Puesto que `S` consiste en dos señales de entrada numeradas 0 y 1, el índice para `S` irá desde 0 hasta 1 (`0 to 1`). Los componentes de este vector son `S(0)` y `S(1)`. `I` consiste en cuatro señales de entrada numeradas desde 0 hasta 3, por lo que el índice irá desde 0 hasta 3 (`0 to 3`). Igualmente, en las líneas 24 y 25, especificamos las señales `S_n`, `D`, y `N` como vectores del tipo *std_logic_vector*. `D` representa las salidas decodificadas, y `N` representa las cuatro señales internas entre las puertas AND y las puertas OR.

```
-- Multiplexor 4-a-1: descripción VHDL estructural                    -- 1
-- (véase la Figura 4-14 para el diagrama lógico)                     -- 2
library ieee, lcdf_vhdl;                                              -- 3
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;                -- 4
entity multiplexer_4_to_1_st is                                       -- 5
  port (S: in std_logic_vector(0 to 1);                               -- 6
        I: in std_logic_vector(0 to 3);                               -- 7
        Y: out std_logic);                                            -- 8
end multiplexer_4_to_1_st;                                            -- 9
                                                                      --10
architecture structural_2 of multiplexer_4_to_1_st is                 --11
  component NOT1                                                      --12
    port(in1: in std_logic;                                           --13
         out1: out std_logic);                                        --14
  end component;                                                      --15
  component AND2                                                      --16
    port(in1, in2: in std_logic;                                      --17
         out1: out std_logic);                                        --18
  end component;                                                      --19
  component OR4                                                       --20
    port(in1, in2, in3, in4: in std_logic;                            --21
         out1: out std_logic);                                        --22
  end component;                                                      --23
  signal S_n: std_logic_vector(0 to 1);                               --24
  signal D, N: std_logic_vector(0 to 3);                              --25
  begin                                                               --26
    g0: NOT1 port map (S(0), S_n(0));                                 --27
    g1: NOT1 port map (S(1), S_n(1));                                 --28
    g2: AND2 port map (S_n(1), S_n(0), D(0));                         --29
    g3: AND2 port map (S_n(1), S(0), D(1));                           --30
    g4: AND2 port map (S(1), S_n(0), D(2));                           --31
    g5: AND2 port map (S(1), S(0), D(3));                             --32
    g6: AND2 port map (D(0), I(0), N(0));                             --33
    g7: AND2 port map (D(1), I(1), N(1));                             --34
    g8: AND2 port map (D(2), I(2), N(2));                             --35
    g9: AND2 port map (D(3), I(3), N(3));                             --36
    g10: OR4 port map (N(0), N(1), N(2), N(3), Y);                    --37
  end structural_2;                                                   --38
```

□ **FIGURA 4-28**
Descripción estructural VHDL de un multiplexor 4 a 1

Empezando en la línea 27, observe cómo se referencian las señales del tipo *std_logic_vector* empleando el nombre de la señal y el índice entre paréntesis. También es posible referirse a un subvector (por ejemplo, `N(1 to 2)` que se refiere a `N(1)` y `N(2)`, las señales centrales de `N`). También, si se desea que el índice mayor aparezca primero, VHDL usa una notación algo diferente. Por ejemplo, `signal N: std_logic_vector (3 downto 0)` define el primer bit de la señal `N` como `N(3)` y el último como `N(0)`.

En las líneas de la 27 a 37, se emplea un método alternativo para especificar el mapeado de los puertos para las puertas lógicas. En lugar de dar explícitamente los nombres de las entradas y las salidas del componente, asumimos que estos nombres están en el mapeado del puerto en el mismo orden que el dado para el componente. Entonces podemos especificar implícitamente las señales ligadas a estos nombres listándolas en el mismo orden que dichos nombres. Por ejem-

plo, en la línea 29, `S_n(1)` aparece primero, y por tanto, se conecta a `in1`. `S_n(0)` aparece segundo, por lo que se conecta a `in2`. Finalmente, `D(0)` se conecta a `out1`.

Por otra parte, esta descripción VHDL es similar en estructura a la empleada para el decodificador 2 a 4, excepto que el esquemático representado es el de la Figura 4-14. ■

DESCRIPCION DE FLUJO DE DATOS Una descripción de flujo de datos (*dataflow*) describe un circuito en términos de su función en lugar de su estructura y se lleva a cabo mediante sentencias de asignación concurrentes o sus equivalentes. Las sentencias de asignación concurrentes se ejecutan concurrentemente (es decir, en paralelo) siempre que alguno de los valores del lado derecho de la sentencia varía. Por ejemplo, siempre que ocurra un cambio en un valor en el lado derecho de una ecuación booleana, se evalúa el lado izquierdo. El uso de descripciones de flujo de datos compuesto por ecuaciones booleanas se ilustra en el Ejemplo 4-17.

EJEMPLO 4-17 Flujo de datos VHDL para un decodificador 2 a 4

La Figura 4-29 muestra una descripción VHDL del circuito decodificador 2 a 4 de la Figura 4-10. Este ejemplo se usará para mostrar una descripción de flujo de datos compuesta por ecuaciones booleanas. La librería a usar y la declaración de la entidad son idénticas a las de la Figura 4-27, por lo que no se repetirán aquí. La descripción del flujo de datos empieza en la línea 9. Las señales `A0_n` y `A1_n` se definen mediante asignaciones de señal que aplican el operador **`not`** a la señal de entrada `A0` y `A1`, respectivamente. En la línea 11, `A0_n`, `A1_n` y `EN` se combinan con un operador `and` para formar `D0`. De forma similar se definen `D1`, `D2` y `D3` en las líneas 12 a 14. Véase cómo esta descripción de flujo de datos es mucho más simple que la descripción estructural dada en la Figura 4-27.

```
-- Decodificador 2-a-4: descripción de flujo de datos VHDL                -- 1
-- (véase la Figura 4-10 para el diagrama lógico)                          -- 2
Use library, use, and entity entries from 2_to_4_decoder_st;               -- 3
                                                                           -- 4
architecture dataflow_1 of decoder_2_to_4_w_enable is                      -- 5
                                                                           -- 6
signal A0_n, A1_n: std_logic;                                              -- 7
begin                                                                      -- 8
  A0_n <= not A0;                                                          -- 9
  A1_n <= not A1;                                                          -- 10
  D0 <= A0_n and A1_n and EN;                                              -- 11
  D1 <= A0 and A1_n and EN;                                                -- 12
  D2 <= A0_n and A1 and EN;                                                -- 13
  D3 <= A0 and A1 and EN;                                                  -- 14
end dataflow_1;                                                            -- 15
```

□ **FIGURA 4-29**
Descripción de flujo de datos VHDL de un decodificador 2 a 4 ■

En los próximos dos ejemplos, describimos el multiplexor 4 a 1 para ilustrar dos formas alternativas de descripción de flujo de datos: *when-else* y *with-select*.

EJEMPLO 4-18 VHDL de un multiplexor 4 a 1 usando *when-else*

En la Figura 4-30, se emplea, en vez de sentencias en forma de ecuaciones booleanas, la sentencia *when-else* para describir la arquitectura del multiplexor. Esta sentencia representa la tabla

```
-- Multiplexor 4-a-1: descripción VHDL de flujo de datos condicional        --  1
-- empleando when-else (ver la Tabla 4-7 para la tabla de funcionamiento)   --  2
library ieee;                                                               --  3
 se ieee.std_logic_1164.all;                                                --  4
entity multiplexer_4_to_1_we is                                             --  5
  port (S : in std_logic_vector(1 downto 0);                                --  6
        I : in std_logic_vector(3 downto 0);                                --  7
        Y : out std_logic);                                                 --  8
end multiplexer_4_to_1_we;                                                  --  9
                                                                            -- 10
architecture function_table of multiplexer_4_to_1_we is                     -- 11
begin                                                                       -- 12
  Y <= I(0) when S = "00" else                                              -- 13
       I(1) when S = "01" else                                              -- 14
       I(2) when S = "10" else                                              -- 15
       I(3) when S = "11" else                                              -- 16
       'X';                                                                 -- 17
end function_table;                                                         -- 18
```

□ **FIGURA 4-30**
Descripción VHDL de flujo de datos condicional VHDL de un multiplexor 4 a 1 usando *when-else*

de verdad de la función dada en la Tabla 4-7. Cuando `S` toma un determinado valor binario entonces se asigna entrada `I(i)` concreta a la salida `Y`. Cuando el valor de `S` es `00`, entonces `I(0)` se asigna a `Y`. En otros casos se invoca el **`else`** y cuando el valor de `S` es `01`, entonces se asigna `I(1)` a `Y`, y así sucesivamente. En *std_logic*, cada bit puede tomar 9 valores diferentes. Así que el par de bits para `S` puede tomar 81 posibles valores, pero hasta ahora sólo se han especificado 4 de ellos. Con el fin de definir `Y` para los 77 valores restantes, se coloca la última sentencia **`else`** seguida por `X` (desconocida). Esto asigna a `Y` el valor `X`, para cualquiera de los 77 valores restantes que se den en `S`. Este valor de la salida sólo se genera durante la simulación, sin embargo, en el circuito real, `Y` siempre valdrá 0 o 1. ■

EJEMPLO 4-19 VHDL de un multiplexor 4 a 1 usando *with-select*

Whith-select es una variación del *when-else*, como se ilustra, para el multiplexor 4 a 1, en la Figura 4-31. La expresión cuyo valor se empleará para tomar la decisión se coloca detrás de **`with`** y antes de **`select`**. Los valores de la expresión que causan las distintas asignaciones siguen a **`when`** y se separan unos de otros mediante comas. En el ejemplo, `S` es la señal que determina cuál será el valor seleccionado para `Y`. Cuando `S="00"`, `I(0)` se asigna a `Y`. Cuando `S="01"`, `I(1)` se asigna `Y` y así sucesivamente. `'X'` se asigna a `Y` **`when others`**, donde **`others`** (otros) representa las 77 combinaciones *std_logic* no especificadas hasta ahora. ■

Observe que *when-else* permite tomar decisiones sobre varias señales distintas. Por ejemplo, para el demultiplexor de la Figura 4-10, el primer **`when`** puede condicionarse en la entrada `EN` y los siguientes **`when`** sobre la entrada `S`. Por contra, el *with-select* sólo puede depender de una única condición booleana (por ejemplo, `EN` o `S`, pero no ambas). También es normal que en las herramientas típicas de síntesis, el *when-else* se implemente con una estructura lógica más compleja, puesto que cada una de las decisiones depende no sólo de la condición que se está evaluando actualmente, sino también de todas las decisiones anteriores. Como consecuencia, la estructura que se sintetiza tiene en cuenta este orden de prioridad, reemplazando la puerta

```
-- Multiplexor 4-a-1: descripción VHDL de flujo de datos condicional       --  1
-- empleando with Select (ver la Tabla 4-7 para la tabla de funcionamiento) --  2
library ieee;                                                              --  3
 se ieee.std_logic_1164.all;                                               --  4
entity multiplexer_4_to_1_ws is                                            --  5
  port (S : in std_logic_vector(1 downto 0);                               --  6
        I : in std_logic_vector(3 downto 0);                               --  7
        Y : out std_logic);                                                --  8
end multiplexer_4_to_1_ws;                                                 --  9
                                                                           -- 10
architecture function_table_ws of multiplexer_4_to_1_ws is                 -- 11
begin                                                                      -- 12
  with S select                                                            -- 13
    Y <= I(0) when "00",                                                   -- 14
         I(1) when "01",                                                   -- 15
         I(2) when "10",                                                   -- 16
         I(3) when "11",                                                   -- 17
         'X' when others;                                                  -- 18
end function_table_ws;                                                     -- 19
```

□ **FIGURA 4-31**
Descripción VHDL de flujo de datos condicional de un multiplexor 4 a 1 usando *with-select*

4 × 2 AND-OR por una cadena de cuatro multiplexores 2 a 1. Por contra no hay ninguna dependencia directa entre las decisiones de *with-select*. *With-select* se implementa con un decodificador y la puerta 4 × 2 AND-OR.

Ya hemos cubierto muchos de los fundamentos de VHDL que se necesitan para describir circuitos combinacionales. Continuaremos con más VHDL presentando las formas de describir circuitos aritméticos en el Capítulo 5 y circuitos secuenciales en el Capítulo 6.

4-8 REPRESENTACIÓN HDL DE CIRCUITOS COMBINACIONALES—VERILOG

Dado que un HDL se emplea para describir y diseñar hardware, es muy importante tener presente cómo se escribe en ese lenguaje el hardware involucrado. Esto es particularmente crítico si la descripción HDL debe ser sintetizada. Por ejemplo, si se ignora el hardware que se generará, es muy fácil especificar una estructura de puertas enorme y compleja usando × (multiplicación) cuando todo lo que se hubiera necesitado es una estructura mucho más simple con tan sólo unas pocas puertas. Por esta razón, inicialmente hacemos hincapié en la descripción detallada del hardware con Verilog, y procedemos después a descripciones más abstractas de niveles superiores.

Los ejemplos seleccionados en este capítulo son útiles para introducir Verilog como un medio alternativo para representar detalladamente circuitos digitales. Inicialmente mostramos las descripciones estructurales Verilog que sustituyen al esquemático para el decodificador 2 a 4 con habilitación de la Figura 4-10. Este ejemplo y otro que usa el multiplexor 4 a 1 de la Figura 4-14, ilustran muchos de los conceptos fundamentales del Verilog. Después presentaremos descripciones Verilog de nivel funcional superior y descripciones Verilog de comportamiento para estos circuitos que ilustrarán más conceptos fundamentales de Verilog.

EJEMPLO 4-20 Verilog estructural para un decodificador 2 a 4

La Figura 4-32 muestra una descripción Verilog para el circuito decodificador 2 a 4 de la Figura 4-10, en la página 152. Este ejemplo se usará para mostrar varias características generales del Verilog así como la descripción estructural de circuitos.

El texto entre dos `//` y el final de línea se interpreta como un comentario, así que la descripción en la Figura 4-27 empieza con dos líneas de comentarios. Para comentarios de varias líneas existe una notación alternativa que emplea `/` y `*`:

```
/* decodificador 2 a 4 con habilitación: descripción estructural Verilog
// (ver Figura 4-10 para el diagrama lógico) */
```

Para ayudar en la discusión de esta descripción Verilog se han agregado comentarios, a la derecha, que proporcionan los números de línea. Como lenguaje que es, Verilog tiene una sintaxis que describe de forma precisa las estructuras válidas que pueden usarse. Este ejemplo ilustrará muchos aspectos de la sintaxis. En particular véase el uso de comas y puntos y comas en la descripción. Las comas (`,`) se emplean normalmente para separar elementos de una lista y los puntos y comas (`;`) para terminar las sentencias Verilog.

La línea 3 comienza con la declaración de un **`module`** (módulo) que es la unidad fundamental de un diseño Verilog. El resto de esta descripción, hasta el **`endmodule`** de la línea 20, define el módulo. Note que no hay `;` después de **`endmodule`**. Al igual que en el símbolo de un esquemático necesitamos darle un nombre al diseño y definir sus entradas y salidas. Esta es la función de la *declaración de módulo* de la línea 3 y de las sentencias *input* y *output* que siguen. En Verilog, **`module`**, **`input`** y **`output`** son palabras claves. Las palabras clave, que nosotros mostramos en tipo negrita, tienen un significado especial y no pueden usarse para nombrar objetos tales como los módulos, entradas, salidas o cables. La sentencia **`module`** `decoder_2_to_4_st_v` declara que existe un diseño (o parte del diseño) con el nombre `decoder_2_to_4_st_v`.

```
// Decodificador 2-a-4 con habilitación: descripción estructural Verilog    //  1
// (véase Figura 4-10 para el diagrama lógico)                               //  2
module decoder_2_to_4_st_v(EN, A0, A1, D0, D1, D2, D3);                      //  3
  input EN, A0, A1;                                                          //  4
  output D0, D1, D2, D3;                                                     //  5
                                                                             //  6
  wire A0_n, A1_n, N0, N1, N2, N3;                                           //  7
  not                                                                        //  8
    go(A0_n, A0),                                                            //  9
    g1(A1_n, A1);                                                            // 10
  and                                                                        // 11
    g3(N0, A0_n, A1_n),                                                      // 12
    g4(N1, A0, A1_n),                                                        // 13
    g5(N2, A0_n, A1),                                                        // 14
    g6(N3, A0,A1),                                                           // 15
    g7(D0, N0, EN),                                                          // 16
    g8(D1, N1, EN),                                                          // 17
    g9(D2, N2, EN),                                                          // 18
    g10(D3, N3, EN);                                                         // 19
endmodule                                                                    // 20
```

□ **FIGURA 4-32**
Descripción estructural en Verilog de un decodificador 2 a 4

Verilog es sensible a las mayúsculas (es decir, los nombres y palabras claves se distinguen por estar escritas en letras mayúsculas o minúsculas). `DECODER_2_4_st_v` es distinto de `Decoder_2_4_st_v` o `decoder_2_4_st_V`.

Tal y como se haría con un símbolo de un esquemático, nombramos las entradas y salidas del decodificador en la declaración del módulo. Después, se usará una *declaración de entradas* para definir qué nombres en la declaración del módulo son entradas. En el diseño del ejemplo hay tres señales de entrada `EN`, `A0`, y `A1`. El hecho de que éstas son las entradas se denota por el uso de la palabra clave **`input`**. De manera similar se emplea una *declaración de salidas* para definir las salidas, que en este caso se nombran como `D0`, `D1`, `D2` y `D3` y se distinguen como salidas por el empleo de la palabra clave **`output`**.

Las entradas y salidas así como otros tipos de señales binarias en Verilog pueden tomar uno de cuatro valores distintos. Los dos valores obvios son 0 y 1. Se añade la `x` para representar valores desconocidos y `z` para representar la alta impedancia en las salidas de 3 estados. Verilog también tiene valores de fuerza que, cuando se combinan con los cuatro valores dados, proporcionan 120 posibles estados para las señales. Los valores de fuerza se usan en el modelado de circuitos electrónicos, sin embargo, aquí no se tendrán en cuenta.

DESCRIPCIÓN ESTRUCTURAL Ahora, especificaremos la función del decodificador. En este caso, usamos una *descripción estructural* que es equivalente al esquemático mostrado en la Figura 4-10. Observe que el esquemático está hecho con puertas. Verilog proporciona 14 puertas primitivas como palabras claves. De éstas, ahora sólo estamos interesados en ocho: **`buf`**, **`not`**, **`and`**, **`or`**, **`nand`**, **`nor`**, **`xor`** y **`xnor`**. **`buf`** y **`not`** tiene una única entrada y todos los otros tipos de puertas pueden tener de dos a cualquier número entero de entradas. **`buf`** es un buffer que implementa la función $z = x$, con x como la entrada y z como la salida. Es un amplificador de señales electricas que puede usarse para conseguir un mayor *fan-out* o menores retardos. **`xor`** es la puerta OR-exclusiva y **`xnor`** es la puerta del NOR-exclusiva, el complemento de la OR-exclusiva. En nuestro ejemplo, usaremos simplemente dos tipos de puertas, **`not`** y **`and`** como se muestra en las líneas 8 y 11 de la Figura 4-32.

Antes de especificar la interconexión de las puertas, que es lo mismo que el *netlist* del circuito, hay que nombrar todos los nodos del circuito. Las entradas y salidas ya tienen nombres. Los nodos interiores son las salidas de los dos inversores y de las cuatro puertas AND más a la izquierda en la Figura 4-10. En la línea 7, estos nodos se declaran como *cables* mediante el empleo de la palabra clave **`wire`**. Los nombres `A0_n` y `A1_n` se emplean para las salidas del inversor y `N0`, `N1`, `N2` y `N3` para las salidas de las puertas AND. En Verilog, **`wire`** es el tipo predefinido para un nodo. El tipo predefinido para los puertos **`input`** y **`output`** es **`wire`**.

Siguiendo a la declaración de las señales internas, el circuito descrito contiene dos inversores y ocho puertas AND de 2-entradas. Una declaración consiste en un tipo de puerta seguido por una lista de instancias de ese tipo de puerta separada por comas. Cada instancia consiste en el nombre de la puerta y, entre paréntesis, la salida y las entradas de la puerta separadas por comas, con la salida colocada en primer lugar. La primera declaración empieza en la línea 8 con el tipo de puerta **`not`**. Seguido está el inversor `g0` con `A0_n` como salida y `A0` como entrada. Para completar la declaración, el inversor `g1` se describe de forma similar. Desde la línea 11 hasta la 19 se definen las ocho puertas restantes y se indican cómo están conectadas las señales a sus salidas y entradas. Por ejemplo, en la línea 14, se define el caso de una puerta AND de 2 entradas nombrada como `g5`. Tiene como salida `N2` y como entradas `A0_n` y `A1`. El módulo se completa con la palabra clave **`endmodule`**. ■

EJEMPLO 4-21 Verilog estructural de un multiplexor 4 a 1

En la Figura 4-33 se muestra la descripción estructural del multiplexor 4 a 1 de la Figura 4-14, que sirve para ilustrar el concepto Verilog de vector. En las líneas 4 y 5, en lugar de especificar `S` e `I` como cables de un único bit, se definen como cables de varios bits llamados *vectores*. Los bits de un vector son nombrados mediante un rango de enteros. Este rango viene dado por los valores máximo y mínimo. Especificando estos dos valores definimos la anchura del vector y los nombres de cada uno de sus bits. En las líneas 4, 5,8 y 9 se ilustran los rangos de los vectores de la Figura 4-33. La línea **`input`** `[1:0] S` indica que `S` es un vector con una anchura de dos, con el bit de mayor peso numerado `1` y bit menos significativo numerado `0`. Los componentes de `S` son `S[1]` y `S[0]`. La línea **`input`** `[3:0] I` declara `I` como una entrada de 4 bits, con el bit de mayor peso numerado `3` y bit de menor numerado `0`. La línea **`wire`** `[0:3] D` también es un vector de 4 bits que representa los cuatro cables internos entre las puertas AND de la izquierda y de la derecha, pero en este caso, el bit más significativo se numera `0` y el bit menos significativo se numera `3`. Una vez un vector se ha declarado, entonces pueden ser referidos el vector entero o sus subcomponentes. Por ejemplo, `S` se refiere a los dos bits de `S`, y `S[1]` se refiere al bit más significativo de `S`. `N` se refiere a los cuatro bits de `N` y `N[1:2]` se refiere a los bits centrales de `N`. Este tipo de referencias se usa para especificar entradas y salidas al instanciar las puertas, tal y como se aprecia en las líneas 11 a la 25. Por otra parte, esta descripción Verilog es similar en estructura a la del decodificador 2 a 4, excepto que el esquemático representado es el de la Figura 4-14. ■

```
// Multiplexor 4-a-1: descripción Verilog estructural                    //  1
// (véase la Figura 4-14 para el diagrama lógico)                        //  2
module multiplexer_4_to_1_st_v(S, I, Y);                                 //  3
   input [1:0] S;                                                        //  4
   input [3:0] I;                                                        //  5
   output Y;                                                             //  6
                                                                         //  7
   wire [1:0] not_S;                                                     //  8
   wire [0:3] D, N;                                                      //  9
                                                                         // 10
not                                                                      // 11
    gn0(not_S[0], S[0]),                                                 // 12
    gn1(not_S[1], S[1]);                                                 // 13
                                                                         // 14
and                                                                      // 15
   g0(D[0], not_S[1], not_S[0]),                                         // 16
   g1(D[1], not_S[1], S[0]),                                             // 17
   g2(D[2], S[1], not_S[0]),                                             // 18
   g3(D[3], S[1], S[0]);                                                 // 19
   g0(N[0], D[0], I[0]),                                                 // 20
   g1(N[1], D[1], I[1]),                                                 // 21
   g2(N[2], D[2], I[2]),                                                 // 22
   g3(N[3], D[3], I[3]);                                                 // 23
                                                                         // 24
or go(Y, N[0], N[1], N[2], N[3]);                                        // 25
                                                                         // 26
endmodule                                                                // 27
```

□ **FIGURA 4-33**
Descripción estructural en Verilog de un multiplexor 4 a 1

DESCRIPCIÓN DE FLUJO DE DATOS Una descripción de flujo de datos (*dataflow*) es una forma de descripción Verilog que no se basa en la estructura, sino en la función. Una descripción de flujo de datos se realiza a partir de sentencias de flujo de datos. Para la primera descripción de flujo de datos, se usarán ecuaciones Booleanas en lugar de un esquema lógico equivalente. Las ecuaciones Booleanas se ejecutan en el paralelo siempre que cambie cualquiera de los valores del lado derecho de las ecuaciones.

EJEMPLO 4-22 Flujo de datos Verilog de un decodificador 2 a 4

En la Figura 4-34, se da una descripción de flujo de datos para el decodificador 2 a 4. Esta descripción en particular usa una sentencia de asignación consistente en la palabra clave **`assign`** seguida, en este caso, por una ecuación booleana. En estas ecuaciones, usamos los operadores booleanos de bits dados por la Tabla 4-12. En la línea 7 de la Figura 4-34, `EN`, `~A0` y `~A1` se combinan con un operador `&`. Esta combinación `&` se asigna a la salida `D0`. Análogamente, se definen `D1`, `D2` y `D3` en las líneas 8 a 10.

□ **TABLA 4-12**
Operadores Verilog para bits

Operación	Operador
~	NOT
&	AND
|	OR
^	XOR
^~ o ~^	XNOR

```
// Decodificador 2-a-4: con habilitación: flujo de datos Verilog        //  1
// (véase Figura 4-10 para el diagrama lógico)                          //  2
module decoder_2_to_4_df_v(EN, A0, A1, D0, D1, D2, D3);                 //  3
  input EN, A0, A1;                                                     //  4
  output D0, D1, D2, D3;                                                //  5
                                                                        //  6
  assign D0 = EN & ~A1 & ~A0;                                           //  7
  assign D1 = EN & ~A1 & A0;                                            //  8
  assign D2 = EN & A1 & ~A0;                                            //  9
  assign D3 = EN & A1 & A0;                                             // 10
                                                                        // 11
endmodule                                                               // 12
```

□ **FIGURA 4-34**
Descripción de flujo de datos Verilog de un decodificador 2 a 4 ■

En los próximos tres ejemplos describimos el multiplexor 4 a 1 para ilustrar tres formas alternativas de descripción de flujo de datos: las ecuaciones booleanas, las combinaciones binarias como condiciones, y decisiones binarias como condiciones.

EJEMPLO 4-23 Flujo de datos Verilog de un multiplexor 4 a 1

En la Figura 4-35, una única ecuación booleana para `Y` describe el multiplexor. Esta ecuación está en forma de suma-de-productos con `&` para AND y `|` para OR. Como variables se emplean componentes de los vectores `S` e `I`.

```
// Multiplexor 4-a-1: flujo de datos Verilog
// (véase Figura 4-14 para el diagrama lógico)
module multiplexer_4_to_1_df_v(S, I, Y);
  input [1:0] S;
  input [3:0] I;
  output Y;

  assign Y = (~ S[1] & ~ S[0] & I[0]) | (~ S[1] & S[0] & I[1])
           | (S[1] & ~ S[0] & I[2]) | (S[1] & S[0] & I[3]);
endmodule
```

□ **FIGURA 4-35**
Descripción de flujo de datos Verilog de un multiplexor 4 a 1 usando una ecuación booleana ■

EJEMPLO 4-24 Verilog de un multiplexor 4 a 1 usando combinaciones

La descripción de la Figura 4-36 es equivalente a la tabla de la función dada en la Tabla 4-7 de la página 148 usando un operador condicional sobre las combinaciones binarias. Si el valor lógico entre paréntesis es cierto, entonces el valor antes de : se asigna a la variable independiente, en este caso, `Y`. Si el valor lógico es falso, entonces se asigna el valor que hay después de `:`. El operador de igualdad lógica se denota por `==`. Suponga que consideramos la condición `S==2'b00. 2'b00` representa una constante. El `2` especifica que la constante contiene dos dígitos, `b` que la constante se da en binario, y `00` proporciona el valor de la constante. Así, la expresión tiene el valor verdadero si el vector `S` es igual a `00`; de lo contrario, es falso. Si la expresión es verdadera, entonces `I[0]` se asigna a `Y`. Si la expresión es falsa, entonces se evalúa la siguiente expresión conteniendo `?`, y así sucesivamente. En este caso, para que una condición sea evaluada, todas las condiciones que la preceden deben evaluarse como falsas. Si ninguna de las decisiones se evalúa como verdadera, entonces el valor por defecto `1'bx` se asigna a `Y`. Recuerde que el valor por defecto `x` significa «desconocido».

```
// Multiplexor 4-a-1: flujo de datos Verilog
// (véase Tabla 4-7 para la tabla de funcionamiento)
module multiplexer_4_to_1_cf_v(S, I, Y);
  input [1:0] S;
  input [3:0] I;
  output Y;

  assign Y = (S == 2'b00) ? I[0] :
             (S == 2'b01) ? I[1] :
             (S == 2'b10) ? I[2] :
             (S == 2'b11) ? I[3] : 1'bx ;
endmodule
```

□ **FIGURA 4-36**
Descripción Verilog de flujo de datos condicional de un multiplexor 4 a 1 usando combinaciones ■

EJEMPLO 4-25 Verilog de un multiplexor 4 a 1 usando decisiones binarias

La última forma de descripción de flujo de datos se muestra en la Figura 4-37. Está basada en el empleo de operadores condicionales para formar un árbol de decisión que se corresponda con

una expresión booleana factorizada. En este caso, si `S[1]` es `1`, entonces `S[0]` se evalúa para determinar si a `Y` se asigna `I[3]` o `I[2]`. Si `S[1]` es `0`, entonces `S[0]` se evalúa para determinar si a `Y` se asigna `I[1]` o `I[0]`. Para una estructura regular como un multiplexor, esta aproximación basada en decisiones de dos caminos (binarias) obtiene una expresión muy simple para la descripción del flujo de datos.

```
// Multiplexor 4-a-1: flujo de datos Verilog
// (véase Tabla 4-7 para la tabla de funcionamiento)
module multiplexer_4_to_1_tf_v(S, I, Y);
  input [1:0] S;
  input [3:0] I;
  output Y;

  assign Y = S[1] ? (S[0] ? I[3] : I[2]) :
             (S[0] ? I[1] : I[0]) ;
endmodule
```

□ **FIGURA 4-37**
Descripción Verilog de flujo de datos condicional para un multiplexor 4 a 1 usando decisiones binarias ■

Esto completa la introducción a Verilog. Continuaremos con más Verilog presentando las formas de describir los circuitos aritméticos en el Capítulo 5 y los circuitos secuenciales en el Capítulo 6.

4-9 RESUMEN DEL CAPÍTULO

En este capítulo se han tratado varios tipos de circuitos combinacionales, denominados frecuentemente bloques funcionales y que se usan para diseñar circuitos más grandes. Se introdujeron circuitos básicos que llevan a cabo funciones de una sola variable. Se ha tratado el diseño de decodificadores que activan una de sus varias líneas de salida en respuesta a un código de entrada. Los codificadores, a la inversa de los decodificadores, generan un código asociado a la línea activa de un conjunto de líneas de entrada. Se ha ilustrado el diseño de multiplexores que toman datos aplicados a la entrada seleccionada y los entregan a la salida.

Se ha abordado el diseño de circuitos lógicos combinacionales usando decodificadores, multiplexores, y lógica programable. En combinación con puertas OR, los decodificadores proporcionan una aproximación sencilla, basada en mini términos, para realizar circuitos combinacionales. Se han estudiado procedimientos para implementar cualquier función booleana de n entradas a partir de un multiplexor n a 1 o a partir de un inversor y un multiplexor $(n - 1)$ a 1. Las memorias ROM pueden programarse para que contengan tablas de verdad. PLAs y PALs pueden programarse mediante tablas de programación especializadas. La descomposición y extracción lógica de múltiples niveles del Capítulo 2 permite el mapeado de ecuaciones combinacionales para su implementación mediante tablas de búsqueda.

Las últimas dos secciones del capítulo introdujeron la descripción de circuitos combinacionales en VHDL y Verilog. Cada uno de los HDLs se ilustró mediante descripciones a nivel estructural, funcional, y de comportamiento para varios bloques funcionales presentados con anterioridad en el capítulo.

REFERENCIAS

1. MANO, M. M.: *Digital Design*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2002.
2. WAKERLY, J. F.: *Digital Design: Principles and Practices*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2000.
3. *High-Speed CMOS Logic Data Book*. Dallas: Texas Instruments, 1989.
4. *IEEE Standard VHDL Language Reference Manual*. (ANSI/IEEE Std 1076-1993; revision of IEEE Std 1076-1987). New York: The Institute of Electrical and Electronics Engineers, 1994.
5. SMITH, D. J.: *HDL Chip Design*. Madison, AL: Doone Publications, 1996.
6. PELLERIN, D. and D. TAYLOR: *VHDL Made Easy!* Upper Saddle River, NJ: Prentice Hall PTR, 1997.
7. STEFAN, S. and L. LINDH: *VHDL for Designers*. London: Prentice Hall Europe, 1997.
8. YALAMANCHILI, S.: *VHDL Starter's Guide*. Upper Saddle River, NJ: Prentice Hall, 1998.
9. *IEEE Standard Description Language Based on the Verilog(TM) Hardware Description Language* (IEEE Std 1364-1995). New York: The Institute of Electrical and Electronics Engineers, 1995.
10. PALNITKAR, S.: *Verilog HDL: A Guide to Digital Design and Synthesis.* Upper Saddle River, NJ: SunSoft Press (A Prentice Hall Title), 1996.
11. BHASKER, J.: *A Verilog HDL Primer*. Allentown, PA: Star Galaxy Press, 1997.
12. THOMAS, D., and P. MOORBY: *The Verilog Hardware Description Language* 4th ed. Boston: Kluwer Academic Publishers, 1998.
13. CILETTI, M.: *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL*, Upper Saddle River, NJ: Prentice Hall, 1999.

PROBLEMAS

El símbolo (+) indica problemas más avanzados y el asterisco (*) indica que la solución se puede encontrar en el sitio web del libro: http://www.librosite.net/Mano.

4-1. ***(a)** Dibuje un diagrama que implemente la función constante $F = (F_7, F_6, F_5, F_4, F_3, F_2, F_1, F_0) = (1, 0, 0, 1, 0, 1, 1, 0)$ usando los símbolos de masa y alimentación de la Figura 4-2(b).

(b) Dibuje un diagrama que implemente la función $G = (G_7, G_6, G_5, G_4, G_3, G_2, G_1, G_0) = (A, \bar{A}, 0, 1, \bar{A}, A, 1, 1)$ usando las entradas 1, 0, A y $\bar{A}$.

4-2. **(a)** Dibuje un diagrama que implemente la función $F = (F_7, F_6, F_5, F_4, F_3, F_2, F_1, F_0) = (1, 0, A, \bar{A}, \bar{A}, A, 0, 1)$, usando los símbolos de masa y tensión de alimentación de la Figura 4-2(b) y el cable y el inversor de la Figura 4-2(c) y 4-2(d).

(b) Dibuje un diagrama que implemente la función $G = (G_7, G_6, G_5, G_4, G_3, G_2, G_1, G_0) = (F_3, \bar{F}_2, 0, 1, \bar{F}_1, F_0, 1, 1)$, usando los símbolos de masa y tensión de alimentación y los componentes de vector F.

4-3. **(a)** Dibuje un diagrama que implemente $G = (G_7, G_6, G_5, G_4, G_3, G_2, G_1, G_0) = (F_{11}, F_{10}, F_9, F_8, F_3, F_2, F_1, F_0)$.

(b) Dibuje una implementación sencilla de $H = (H_7, H_6, H_5, H_4, H_3, H_2, H_1, H_0) = (F_3, F_2, F_1, F_0, G_3, G_2, G_1, G_0)$.

4-4. Un sistema de seguridad doméstico tiene un interruptor principal que se usa para habilitar una alarma, luces, cámaras de video, y una llamada a la policía local en el caso de que uno o más de seis juegos de sensores detecte a un intruso. Las entradas, salidas, y las operaciones de habilitación lógicas se especifican a continuación:

Entradas:

S_i, $i = 0, 1, 2, 3, 4, 5$ - señales de seis juegos de sensores (0 - detectado intruso, 1 - ningún intruso)

P - el interruptor principal (0 - sistema de seguridad encendido, 1 - sistema de seguridad apagado)

Salidas:

A - alarma (0 - alarma encendida, 1 - alarma apagada)

L - luces (0 - luces encendidas, 1 - luces apagadas)

V - cámaras de video (0 - cámaras apagadas, 1 - cámaras encendidas)

M - llamada a la policía (0 - no llamar, 1 - llamar)

Funcionamiento:

Si uno o más de los juegos de sensores descubre un intruso y el sistema de seguridad esta encendido, entonces todas las salidas están activadas. De lo contrario, todas las salidas estarán apagadas.

Encuentre la implementación con puertas AND, OR e inversores de este circuito de habilitación que minimiza el número total de entradas de puertas.

4-5. Diseñe un decodificador 4 a 16 a partir de dos decodificadores de 3 a 8 y 16 puertas AND de 2-entradas.

4-6. Diseñe un decodificador 4 a 16 con habilitación usando cinco decodificadores 2 a 4 como el de la Figura 4-10.

4-7. *Diseñe un decodificador 5 a 32 usando dos decodificadores 3 a 8 y 32 puertas AND de 2 entradas.

4-8. Se va a diseñar un decodificador especial 4 a 6. Los códigos de la entrada son desde 000 hasta 101. Para un código dado, la salida D_i, con i igual al equivalente decimal del código, es 1 y todas las demás salidas son 0. Diseñe el decodificador empleando un decodificador 2 a 4, un decodificador 1 a 2, y seis puertas AND de 2-entradas, tal que las salidas de todos los decodificadores se usen por lo menos una vez.

4-9. Dibuje el diagrama lógico detallado de un decodificador 3 a 8 que sólo use puertas NOR y NOT. Incluya una entrada de habilitación.

4-10. *Diseñe un codificador con prioridad de 4 entradas con las entradas y salidas como los dados en la Tabla 4-5, pero en el que la entrada D_0 tiene la prioridad más alta y la entrada D_3 tiene la prioridad más baja.

4-11. Obtenga la tabla de verdad de un codificador con prioridad de decimal a binario.

4-12. **(a)** Diseñe un multiplexor 8 a 1 a partir de un decodificador 3- a-8 y una puerta 8 × 2 AND-OR.

(b) Repita la parte (a), usando dos multiplexores 4 a 1 y un multiplexor 2 a 1.

4-13. Diseñe un multiplexor 16 a 1 a partir de un decodificador 4 a 16 y una puerta 16 × 2 AND-OR.

4-14. Diseñe un multiplexor doble 8 a 1 a partir de un decodificador 3 a 8 y dos puertas 8 × 2 AND-OR.

4-15. Diseñe un multiplexor doble 4 a 1 a partir de un decodificador 2 a 4 y ocho buffers de 3 estados.

4-16. Diseñe un multiplexor 8 a 1 a partir de puertas de transmisión.

4-17. Construya un multiplexor 10 a 1 con un decodificador 3 a 8, un decodificador 1 a 2, y una puerta 10 × 3 AND-OR. Los códigos de selección del 0000 al 1001 deben aplicarse directamente a las entradas decodificadoras sin lógica añadida.

4-18. Construya un multiplexor cuádruple 9 a 1 con solo cuatro multiplexores 8 a 1 y un multiplexor cuádruple 2 a 1. Los multiplexores deben interconectarse y las entradas deben etiquetarse de modo que códigos de selección desde 0000 hasta 1000 puedan aplicarse directamente a las entradas de selección del multiplexor sin lógica añadida.

4-19. *Construya un multiplexor 15 a 1 con dos multiplexores 8 a 1. Interconecte los dos multiplexores y etiquete las entradas de modo que se minimice la lógica adicional necesaria para los códigos de selección 0000 hasta 1110.

4-20. Reordene la tabla de verdad condensada del circuito de la Figura 4-10, y verifique que el circuito puede funcionar como un demultiplexor.

4-21. Un circuito combinacional se define por las siguientes tres funciones booleanas:

$$F_1 = \overline{X + Z} + XYZ$$
$$F_2 = \overline{X + Z} + \bar{X}YZ$$
$$F_3 = X\bar{Y}Z + \overline{X + Z}$$

Diseñe el circuito con un decodificador y puertas OR.

4-22. Un circuito combinacional se especifica por las siguientes tres funciones booleanas:

$$F_1(A, B, C) = \Sigma m(0, 3, 4)$$
$$F_2(A, B, C) = \Sigma m(1, 2, 7)$$
$$F_3(A, B, C) = \Pi M(0, 1, 2, 4)$$

Implemente el circuito con un decodificador y puertas OR.

4-23. Implemente un sumador completo con un multiplexor doble 4 a 1 y un único inversor.

4-24. Implemente la función booleana siguiente con un multiplexor 8 a 1 y un único inversor con la variable *D* como entrada:

$$F(A, B, C, D) = \Sigma m(2, 4, 6, 9, 10, 11, 15)$$

4-25. *Implemente la función booleana

$$F(A, B, C, D) = \Sigma m(1, 3, 4, 11, 12, 13, 14, 15)$$

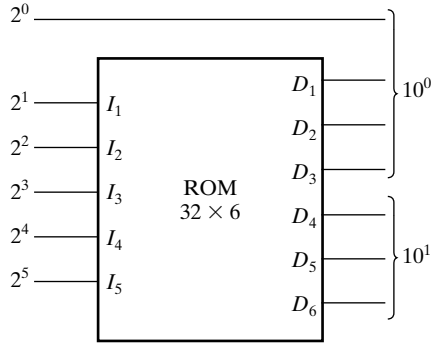
con un multiplexor 4 a 1 y puertas. Conecte las entradas *A* y *B* a las líneas de selección. Las entradas necesarias para las cuatro líneas de datos serán función de las variables *C* y *D*. Los valores de estas variables se obtienen expresando *F* como una función de *C* y *D*

para cada uno de los cuatro casos cuando *AB* igual 00, 01, 10, y 11. Estas funciones implementarse conpuertas.

4-26. Repita el Problema 4-25 usando dos decodificadores 3 a 8 con habilitación, un inversor, y puertas OR con un *fan-in* máximo de 4.

4-27. Dado un chip ROM de 256 × 8 con entrada de habilitación, muestre las conexiones externas necesarias para construir una ROM de 1 K × 16 con ocho chips, un decodificador y puertas OR.

4-28. *La ROM de 32 × 6, junto con la línea 2^0, mostrada en la Figura 4-38 convierte un número binario de 6 bits en su correspondiente número de 2 dígitos BCD. Por ejemplo, el binario 100001 se ha convertido a BCD 011 0011 (decimal 33). Especifique la tabla de verdad de la ROM.



□ **FIGURA 4-38**
Convertidor binario a decimal con ROM

4-29. Especifique el tamaño de una ROM (número de palabras y número de bits por palabra) que almacena la tabla de verdad para los siguientes componentes combinacionales:

(a) Un sumador-restador de 8 bits con C_{in} y C_{out}.
(b) Un multiplicador binario que multiplica dos números del 8-bits.
(c) Un conversor de código BCD a binario para números de 4 dígitos.

4-30. Obtenga la tabla de verdad para una ROM de 8 × 4 que implementa las cuatro funciones booleanas siguientes:

$$A(X, Y, Z) = \Sigma m(0, 1, 2, 6, 7)$$
$$B(X, Y, Z) = \Sigma m(2, 3, 4, 5, 6)$$
$$C(X, Y, Z) = \Sigma m(2, 6)$$
$$D(X, Y, Z) = \Sigma m(1, 2, 3, 5, 6, 7)$$

4-31. Obtenga la tabla de programación del PLA para las cuatro funciones booleanas listadas en el Problema 4-30. Minimice el número de productos. Asegúrese de intentar compartir términos de productos entre funciones que no son implicantes primos de funciones individuales y considere el uso de salidas complementarias (**C**).

4-32. Calcule la tabla de programación del PLA para un circuito combinacional que obtiene el cuadrado de un número de 3 bits. Minimice el número productos.

4-33. Enumere la tabla de programación del PLA para un conversor de código BCD a exceso-3.

4-34. *Repita el Problema 4-33 usando un dispositivo PAL.

4-35. La siguiente es la tabla de verdad de un circuito combinacional de tres-entradas y cuatro-salidas. Obtenga la tabla de programación del PAL para el circuito y marque los fusibles a fundir en un diagrama de la PAL similar al mostrado en Figura 4-26.

Entradas			**Salidas**			
X	***Y***	***Z***	***A***	***B***	***C***	***D***
0	0	0	0	1	0	0
0	0	1	1	1	1	1
0	1	0	1	0	1	1
0	1	1	0	1	0	1
1	0	0	1	0	1	0
1	0	1	0	0	0	1
1	1	0	1	1	1	0
1	1	1	0	1	1	1

Todos los archivos HDL para circuitos referidos en los restantes problemas están disponibles en ASCII para su simulación y edición en la página web del libro. Para los problemas que piden simulación se necesita un compilador/simulador de VHDL o Verilog. En cualquier caso, siempre se pueden escribir las descripciones HDL de muchos problemas sin necesidad de compilar o simular.

4-36. Compile y simule el decodificador 2 a 4 con habilitación de la Figura 4-27 para la secuencia 000, 001, 010, 011, 100, 101, 110, 111 en (`E_n`, `A0`, `A1`). Verifique que el circuito funciona como un decodificador. Primero necesitará compilar la librería `lcdf_vhdl.func_prims` que se usa en la simulación.

4-37. Vuelva a escribir en VHDL el decodificador 2 a 4 dado en la Figura 4-27 que usa (1) la notación `std_logic_vector` en lugar de la notación del `std_logic` para `A` y `D_n` y (2) especificación implícita de los nombres de las entradas y salidas de los componentes por su orden en el paquete `func_prims` de la librería `lcdf_vhdl` dado en el sitio web del libro. *Véase* la Figura 4-28 y el texto que la acompaña para aclarar estos conceptos. Compile y simule el archivo resultante como en el Problema 4-36.

4-38. Compile y simule el multiplexor 4 a 1 de la Figura 4-28 para las 16 combinaciones de 00, 10, 01, 11 en `S` y 1000, 0100, 0010, 0001 en `D`. Primero necesitará compilar la librería `lcdf_vhdl.func_prims` que se usa en la simulación. Verifique que el circuito funciona como un multiplexor.

4-39. *Encuentre un diagrama lógico que se corresponda al VHDL de la descripción estructural de la Figura 4-39. Observe que las entradas complementadas no están disponibles.

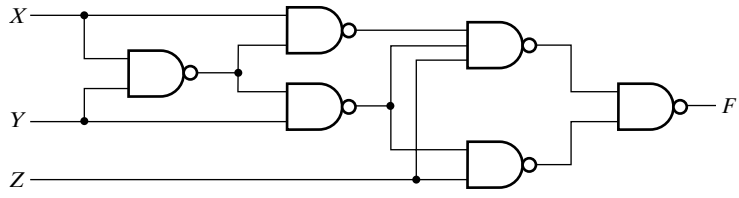
4-40. Empleando la Figura 4-28 como punto de partida escriba una descripción estructural del circuito de la Figura 4-40 en VHDL. Sustituya `X`, `Y` y `Z` por `X(0:2)`. Consulte el paquete `func_prims` de la librería `lcdf_vhdl` para más información sobre los diversos compo-

```
-- Circuito combinacional 1: descripción VHDL estructural
library ieee, lcdf_vhdl;
 se ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;
  entity comb_ckt_1 is
  port(x1, x2, x3, x4 : in std_logic;
       f : out std_logic);
end comb_ckt_1;

architecture structural_1 of comb_ckt_1 is
  component NOT1
    port(in1: in std_logic;
         out1: out std_logic);
  end component;
  component AND2
    port(in1, in2 : in std_logic;
         out1: out std_logic);
  end component;
  component OR3
    port(in1, in2, in3 : in std_logic;
         out1: out std_logic);
  end component;
  signal n1, n2, n3, n4, n5, n6 : std_logic;
  begin
    g0: NOT1 port map (in1 => x1, out1 => n1);
    g1: NOT1 port map (in1 => n3, out1 => n4);
    g2: AND2 port map (in1 => x2, in2 => n1,
                       out1 => n2);
    g3: AND2 port map (in1 => x2, in2 => x3,
                       out1 => n3);
    g4: AND2 port map (in1 => x3, in2 => x4,
                       out1 => n5);
    g5: AND2 port map (in1 => x1, in2 => n4,
                       out1 => n6);
    g6: OR3 port map (in1 => n2, in2 => n5,
                      in3 => n6, out1 => f);
end structural_1;
```

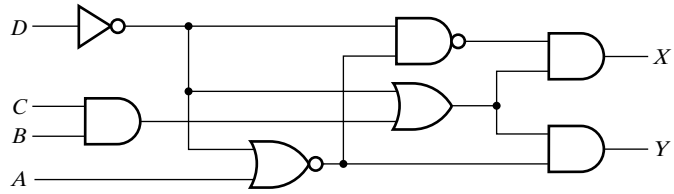
□ **FIGURA 4-39**
VHDL para el Problema 4-39

nentes de la puerta. Compile `func_prims` y su VHDL, y simule su VHDL para las ocho posibles combinaciones de la entrada, verificando la bondad de su descripción.



□ **FIGURA 4-40**
Circuito para los Problemas 4-40, 4-43, 4-51 y 4-53

4-41. Emplee la Figura 4-27 como punto de partida y escriba una descripción estructural en VHDL para el circuito de la Figura 4-41. Consulte el paquete `func_prims` de la librería `lcdf_vhdl` para más información sobre los diversos componentes de la puerta. Compile `func_prims` y su VHDL, y simule su VHDL para las 16 posibles combinaciones de la entrada a fin de verificar la bondad de su descripción.



□ **FIGURA 4-41**
Circuito para los Problemas 4-41 y 4-50

4-42. Encuentre un diagrama lógico que implemente la lógica mínima de dos niveles para la descripción VHDL de flujo de datos de la Figura 4-42. Observe que están disponibles las entradas complementadas.

```
-- Circuito combinacional 2: descripción VHDL de flujo de datos
-library ieee;
 se ieee.std_logic_1164.all;
entity comb_ckt_2 is
  port(a, b, c, d, a_n, b_n, c_n, d_n: in std_logic;
       f, g : out std_logic);
-- a_n, b_n, ... son las negadas de a, b, ... , respectivamente.
end comb_ckt_2;

architecture dataflow_1 of comb_ckt_2 is
begin
  f <= b and (a or (a_n and c)) or (b_n and c and d_n);
  g <= b and (c or (a_n and c_n) or (c_n and d_n));
end dataflow_1;
```

□ **FIGURA 4-42**
VHDL para el Problema 4-42

4-43. *Escriba una descripción de flujo de datos en VHDL para el circuito de la Figura 4-40 usando una ecuación booleana para la salida *F*.

4-44. + Escriba una descripción de flujo de datos en VHDL para el codificador con prioridad de la Figura 4-30 usando *when else*. Compile y simule su descripción con un conjunto de vectores de entrada adecuados para comprobar el correcto funcionamiento del circuito.

4-45. Escriba la descripción de flujo de datos en VHDL para el multiplexor 8 a 1 de la Figu-ra 4-31 usando *with select*. Compile y simule de forma convincente su descripción empleando un conjunto de vectores adecuado a la función del circuito.

4-46. *Compile y simule el decodificador 2 a 4 de la Figura 4-32 para la secuencia 000, 001, 010, 011, 100, 101, 110, 111 en (`E`, `A0`, `A1`). Verifique que el circuito funciona como un decodificador.

4-47. Rescriba la descripción Verilog dada en Figura 4-32 para el decodificador 2 a 4 empleando la notación de vectores para las entradas, salidas, y cables. *Véase* la Figura 4-33 y el texto que la acompaña para recordar estos conceptos. Compile y simule el archivo resultante como en el Problema 4-46.

4-48. Compile y simule el multiplexor 4 a 1 de la Figura 4-33 para las 16 combinaciones de 00, 10, 01, 11 en `S` y 1000, 0100, 0010, 0001 en `D`. Verifique que el circuito funciona como un multiplexor.

4-49. *Encuentre un diagrama lógico que se corresponda con la descripción estructural en Verilog de la Figura 4-43. Observe que las entradas complementadas no están disponibles.

```
// Circuito combinacional 1: descripción Verilog estructural
module comb_ckt_1(x1, x2, x3, x4,f);
  input x1, x2, x3, x4;
  output f;

  wire n1, n2, n3, n4, n5, n6;
  not
    go(n1, x1),
    g1(n4, n3);
  and
    g2(n2, x2, n1),
    g3(n3, x2, x3),
    g4(n5, x3, x4),);
    g5(n6, x1, n4),);
  or
    g6(f, n2, n5, n6),
endmodule
```

□ **FIGURA 4-43**
Verilog para el Problema 4-49

4-50. Emplee la Figura 4-32 como punto de partida y escriba una descripción estructural del circuito de la Figura 4-41 usando Verilog. Compile y simule su Verilog para las 16 posibles combinaciones de la entrada a fin de verificar la bondad de su descripción.

4-51. Emplee la Figura 4-33 como punto de partida y escriba una descripción estructural del circuito de la Figura 4-40 en Verilog. Reemplace `X`, `Y`, y `Z` por **`input`** `[2:0] X`. Compile y simule su Verilog para las ocho posibles combinaciones de la entrada a fin de verificar la bondad de su descripción.

4-52. Encuentre un diagrama lógico que implemente la lógica mínima de dos niveles para la descripción Verilog de flujo de datos de la Figura 4-44. Observe que están disponibles las entradas complementadas.

4-53. *Escriba la descripción de flujo de datos en Verilog para el circuito de la Figura 4-40 usando una ecuación booleana para la salida `F` y la Figura 4-35 como modelo.

4-54. Usando el concepto de flujo de datos condicional de la Figura 4-36 escriba una descripción de flujo de datos Verilog para un multiplexor 8 a 1. Compile y simule su descripción con un juego de vectores de entrada adecuado para la función del circuito.

```
// circuito combinacional 2: flujo de datos Verilog
module comb_ckt_1 (a, b, c, d, a_n, b_n, c_n, d_n, f, g);
// a_n, b_n, ... son las negadas de a, b, ... , respectivamente.
  input a, b, c, d, a_n, b_n, c_n, d_n;
  output f, g;

  assign f = b & (a |(a_n & c)) | (b_n & c & d_n);
  assign g = b & (c | (a_n & c_n) | (c_n & d_n));
endmodule
```

□ **FIGURA 4-44**
Verilog para el Problema 4-52

4-55. + Escriba una descripción de flujo de datos para el codificador con prioridad de la Figura 4-12 usando el concepto de decisión binaria de la Figura 4-37. Compile y simule su descripción con un juego de vectores de entrada adecuados a la funcionalidad del circuito.

CAPÍTULO

5

FUNCIONES Y CIRCUITOS ARITMÉTICOS

En este capítulo, el foco de atención continúa estando en los bloques funcionales, concretamente en una clase especial de bloques funcionales que realizan operaciones aritméticas. Se introduce el concepto de circuito iterativo, realizado a partir de arrays de células combinacionales. Trataremos bloques diseñados como arrays iterativos que realizan sumas, sumas y restas, y multiplicaciones. La simplicidad de estos circuitos aritméticos se logra empleando las representaciones en complemento para los números y la aritmética basada en el complemento. Además, se introduce la contracción de circuitos, que nos permitirá diseñar nuevos bloques funcionales. La contracción supone aplicar asignación de valores a las entradas de los bloques existentes y la simplificación de los circuitos resultantes. Estos circuitos realizan operaciones tales como incrementar o decrementar un número o multiplicar un número por una constante. Muchos de estos nuevos bloques funcionales se usan para construir los bloques funcionales secuenciales del Capítulo 7.

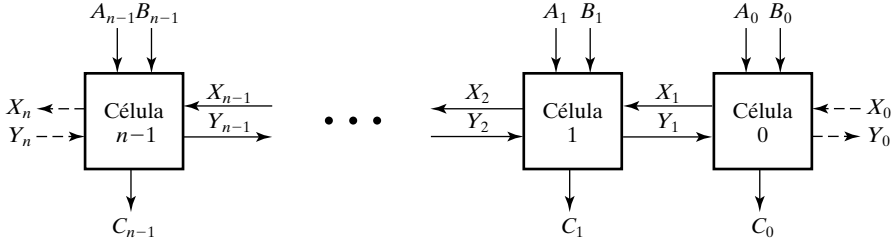
En el diagrama de la computadora genérica que se mostró al comienzo del Capítulo 1, se emplean sumadores, sumadores-restadores, y multiplicadores en el procesador. También se emplean, muy a menudo, incrementadores y decrementadores en otros tantos componentes, en definitiva, los conceptos de este capítulo se aplican a la mayoría de los componentes de la computadora genérica.

5-1 CIRCUITOS COMBINACIONALES ITERATIVOS

En este capítulo, los bloques aritméticos se diseñan típicamente para operar sobre vectores de entrada binarios y producen vectores de salida binarios. Además, la función implementada requiere, en numerosas ocasiones, que una misma función auxiliar se aplique a los bits de cada posición. De este modo, se puede diseñar un bloque funcional para la función auxiliar y entonces este bloque se podrá usar repetidamente en cada posición de bits (o par de bits del mismo peso) del bloque aritmético global a diseñar. A menudo habrá una o más conexiones para pasar los valores entre las posiciones de bits adyacentes. Estas variables interiores son entradas o salidas de la función auxiliar, pero no son accesibles fuera del bloque aritmético global. Los bloques de la función auxiliar se denominan *células* y la implementación global es un *array de células*. Con frecuencia, las células del array son idénticas, aunque no siempre ocurre así. Debido a la naturaleza repetitiva del circuito y a la asociación de un vector de índices a cada una de las células del circuito, el bloque funcional global se denomina *array iterativo*. El empleo de arrays iterativos, un caso especial de circuitos jerárquicos, es útil en el manejo de vectores de bits, por ejemplo, en un circuito que suma dos enteros binarios de 32 bits. Como mínimo, dicho circuito tendría 64 entradas y 32 salidas. En consecuencia, confeccionar las tablas de verdad y escribir las ecuaciones para el circuito entero queda fuera de nuestro alcance. Dado que los circuitos iterativos están basados en células repetitivas, el proceso de diseño se simplifica considerablemente a partir de una estructura básica que guía el diseño.

En la Figura 5-1 se muestra un diagrama de bloques para un circuito iterativo que maneja dos vectores de *n* entradas y produce un vector de *n* salidas. En este caso, hay dos conexiones laterales entre cada par de células del array, una línea de derecha a izquierda y otra de izquierda a derecha. También existen conexiones opcionales, indicadas por líneas discontinuas, en los extremos derechos e izquierdos del array. Una serie arbitraria empleará tantas conexiones laterales como necesite para un diseño en particular. La definición de las funciones asociadas con tales conexiones es muy importante en el diseño del array y de sus células. En particular, el número de conexiones empleadas y sus funciones pueden afectar al coste y a la velocidad de un circuito iterativo.

En la próxima sección, definiremos las células para realizar sumas en posiciones individuales de bits y, a partir de ellas, definiremos un sumador binario como un array de células.



□ **FIGURA 5-1**
Diagrama de bloques de un circuito iterativo

5-2 SUMADORES BINARIOS

Un circuito aritmético es un circuito combinacional que realiza operaciones aritméticas como suma, resta, multiplicación, y división con números binarios o con números decimales en códi-

go binario. Desarrollaremos los circuitos aritméticos mediante diseño jerárquico iterativo. Empezamos en el nivel más bajo encontrando un circuito que realiza la suma de dos dígitos binarios. Esta simple suma consiste en cuatro posibles operaciones: $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$ y $1 + 1 = 10$. Las tres primeras operaciones producen una suma que necesita un único bit para representarla, sin embargo, cuando los dos sumandos son iguales a 1, la suma binaria requiere dos bits. Por ello, en este caso, el resultado se representa siempre por dos bits, el acarreo y la suma. El acarreo que se obtiene de la suma de dos bits se suma al próximo par de bits más significativos. Un circuito combinacional que realiza la suma de dos bits se denomina *semi-sumador*. Un circuito que realiza la suma de tres bits (dos bits significativos y un acarreo anterior) se denomina *sumador completo*. Los nombres de estos circuitos provienen del hecho de que pueden emplearse dos semi-sumadores para implementar un sumador completo. El semi-sumador y el sumador completo son bloques básicos de la aritmética con que se diseñan otros circuitos aritméticos.

Semi-sumador

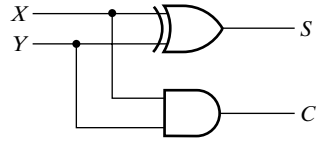
Un semi-sumador es un circuito aritmético que efectúa la suma de dos dígitos binarios. El circuito tiene dos entradas y dos salidas. Las variables de entrada son los sumandos, y por las variables de salida se obtienen la suma y el acarreo. Asignamos los símbolos X e Y a las dos entradas y S (por Suma) y C (por aCarreo, o *Carry*) a las salidas. La tabla de verdad para el semi-sumador se muestra en la Tabla 5-1. La salida C sólo es 1 cuando ambas entradas son 1. La salida S representa el bit menos significativo de la suma. Las funciones booleanas para las dos salidas, obtenidas fácilmente de la tabla de verdad, son:

$$S = \bar{X}Y + X\bar{Y} = X \oplus Y$$
$$C = XY$$

□ **TABLA 5-1**
Tabla de verdad del semi-sumador

Entradas		**Salidas**	
X	***Y***	***C***	***S***
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

El semi-sumador puede implementarse con una puerta OR exclusiva y una puerta AND, tal y como se muestra en la Figura 5-2.



□ **FIGURA 5-2**
Diagrama lógico de un medio sumador

Sumador completo

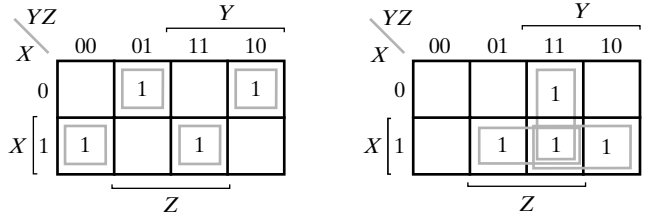
Un sumador completo es un circuito combinacional que efectúa la suma aritmética de tres bits de entrada. Además de las tres entradas, tiene dos salidas. Dos de las variables de entrada, denominadas como *X* e *Y*, representan los dos bits significativos a sumar. La tercera entrada, *Z*, representa el acarreo que procede de la posición anterior menos significativa. Se necesitan dos salidas porque la suma aritmética de tres bits puede tomar valores entre 0 y 3, y el binario de 2 y de 3 necesita dos bits para su representación. De nuevo, las dos salidas se designan por los símbolos *S* para Suma y *C* para aCarreo (o *Carry*); la variable binaria *S* proporciona el valor del bit de la suma, y de la variable binaria *C* se obtiene el acarreo de salida. La tabla de verdad del sumador completo se lista en la Tabla 5-2. Los valores para las salidas se determinan a partir de la suma aritmética de los tres bits de entrada. Cuando todos los bits de las entradas son 0, las salidas son 0. La salida *S* es igual a 1 cuando sólo una entrada es igual a 1 o cuando las tres entradas son iguales a 1. La salida *C* tiene un acarreo de 1 si dos o tres de las entradas son iguales a 1. En la Figura 5-3 se muestran los Mapas de Karnaugh para las dos salidas del sumador completo. Las funciones simplificadas para las dos salidas en forma de suma de productos son:

$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$$

$$C = XY + XZ + YZ$$

□ **TABLA 5-2**
Tabla de verdad del sumador completo

Entradas			Salidas	
X	***Y***	***Z***	***C***	***S***
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$
$= X \oplus Y \oplus Z$

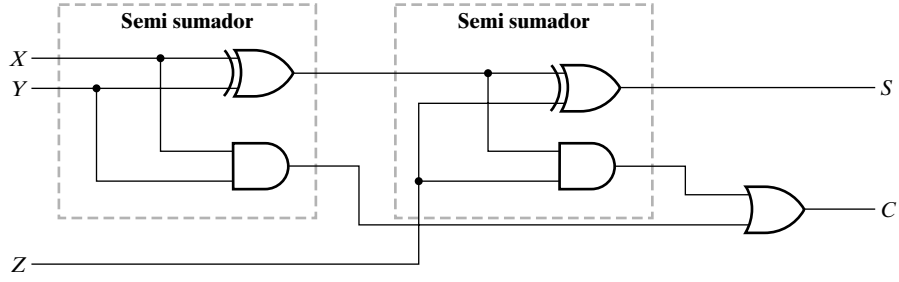
$C = XY + XZ + YZ$
$= XY + Z(X\bar{Y} + \bar{X}Y)$
$= XY + Z(X \oplus Y)$

□ **FIGURA 5-3**
Mapas de Karnaugh para un sumador completo

La implementación a dos niveles requiere de siete puertas AND y dos puertas OR. Sin embargo, el mapa para la salida *S* es idéntico al de un generador de paridad impar, discutida con anterioridad en la Sección 2-7. Además, la función de salida *C* puede manipularse para incluir tres puertas OR exclusivas de *X* e *Y*. Las funciones booleanas resultantes para el sumador completo pueden expresarse en términos de OR exclusivas como:

$$S = (X \oplus Y) \oplus Z$$
$$C = XY + Z(X \oplus Y)$$

El diagrama lógico para esta implementación multinivel, consistente en dos semi-sumadores y una puerta OR, se muestra en la Figura 5-4.

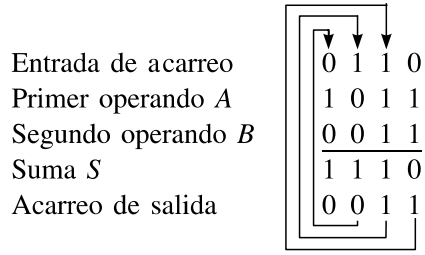


□ **FIGURA 5-4**
Diagrama lógico de un sumador completo

Sumador binario con acarreo serie

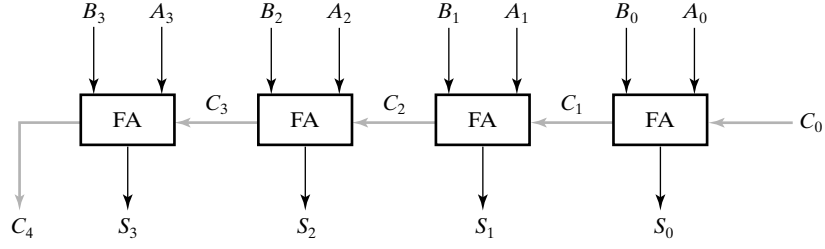
Un sumador binario paralelo es un circuito digital que realiza la suma aritmética de dos números binarios empleando sólo lógica combinacional. El sumador paralelo se construye a partir de *n* sumadores completos en paralelo, donde todos los bits de ambos números a sumar se presentan simultáneamente en paralelo en las entradas para producir la suma. Los sumadores completos se conectan en cascada, con la salida del acarreo de un sumador completo conectada a la entrada de acarreo del siguiente sumador completo. Dado que puede aparecer un acarreo a 1 cerca del bit menos significativo del sumador y propagarse a través de una serie de muchos sumadores completos hacia el bit más significativo, el sumador paralelo se denomina *sumador con acarreo serie* (en inglés *ripple carry adder*)[1]. La Figura 5-5 muestra la interconexión de cuatro bloques sumadores completos para formar un sumador de 4 bits con acarreo serie. Los bits del primer sumando *A* y los del segundo sumando *B* son designados mediante subíndices en orden creciente de derecha a izquierda, de modo que el subíndice 0 denota el bit de menor peso. Los acarreos se conectan en cadena a través de los sumadores completos. El acarreo de entrada del sumador pararelo es C_0, y el acarreo de salida es C_4. Un sumador de n bits con acarreo serie requiere *n* sumadores completos, con cada salida de acarreo conectada a la entrada de acarreo del siguiente sumador completo de orden inmediato superior. Por ejemplo, considere los dos números binarios $A = 1011$ y $B = 0011$. Su suma, $S = 1110$, se forma con un sumador completo de 4 bits con acarreo serie como sigue:

[1] *N. del T.*: El término *ripple*, muchas veces traducido por *rizado*, hace referencia a cómo este bit a 1 se mueve hacía los bits más significativos análogamente al movimiento de la cresta de una ola. Ha parecido más adecuado traducirlo por *acarreo serie*.



El acarreo de entrada para la pareja de bits menos significativa es 0. Cada sumador completo recibe los bits correspondientes de *A* y *B* y la entrada de acarreo y genera el bit de suma *S* y la salida de acarreo. El acarreo de salida de cada posición es el acarreo de entrada de la próxima posición de orden superior, como se indica con las líneas de color.

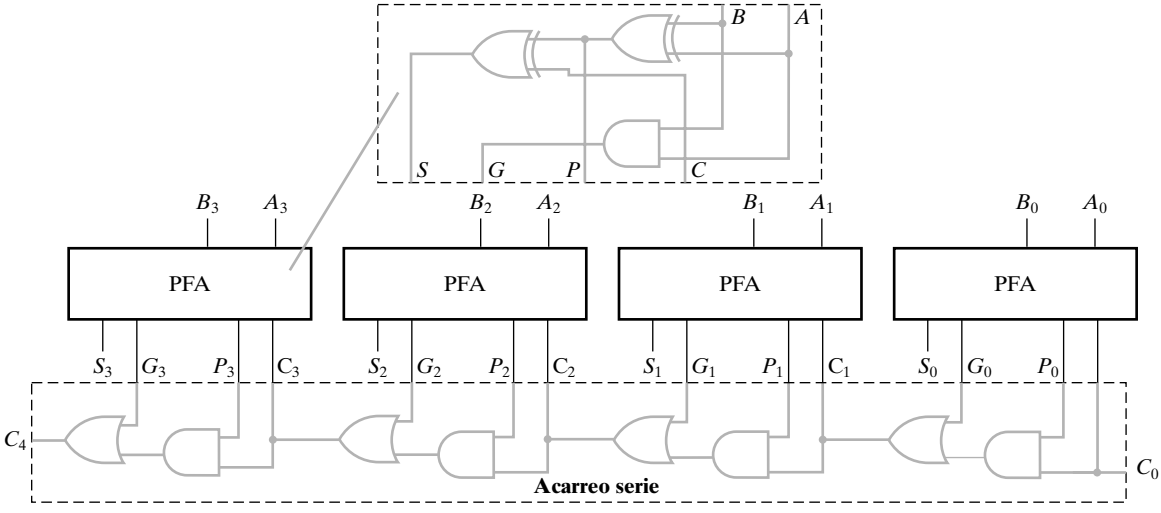
El sumador de 4 bits es un ejemplo típico de un componente digital que puede usarse como un bloque. Puede emplearse en muchas aplicaciones que implican operaciones aritméticas. Observe que el diseño de este circuito por el método usual, y puesto que el circuito presenta 9 entradas, requeriría una tabla de verdad con 512 filas. Colocando directamente cuatro de los sumadores completos conocidos en cascada, es posible obtener una aplicación simple y directa salvando este gran problema. Éste es un ejemplo de la capacidad de los circuitos iterativos y la reutilización de circuitos en el diseño.



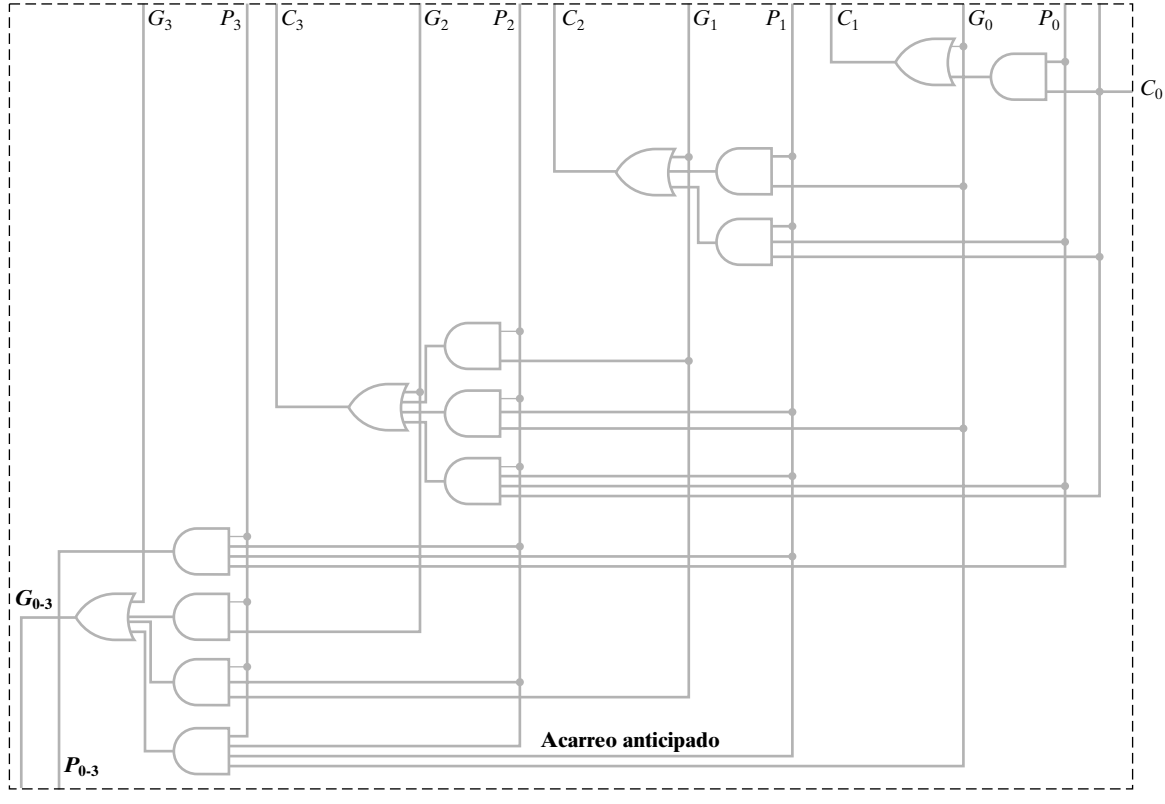
□ **FIGURA 5-5**
Sumador de 4-bits con acarreo serie

Sumador con acarreo anticipado

El sumador con acarreo serie, aunque simple en el concepto, lleva implícito un gran retardo debido a las muchas puertas que hay en el camino del acarreo: del bit menos significativo al bit más significativo. Para un diseño típico, el camino de retardo más largo a través de un sumador de *n*-bits con acarreo serie es $2n + 2$ retardos de puerta. Así, para un sumador de 16 bits con acarreo serie, el retardo es 34 retardos de puerta. Este retardo suele ser uno de los más grandes en un diseño típico para computación. Por ello se ha encontrado un diseño alternativo más atractivo, el *sumador con acarreo anticipado* (o *carry lookahead adder*). Este sumador es un diseño práctico con un retardo reducido, aunque a costa de una mayor complejidad circuital. El diseño para la generación del acarreo se puede obtener mediante una transformación del diseño de acarreo serie consistente en sustituir la lógica del acarreo de los grupos fijos de bits del sumador reducida a lógica de dos niveles. La transformación para un grupo sumador de 4-bits se muestra en la Figura 5-6.



(a)



(b)

□ **FIGURA 5-6**
Desarrollo de un sumador con generación de acarreo

Primero, construimos una nueva jerarquía lógica, separando de los sumadores completos la parte que no está involucrada en el camino de propagación del acarreo de aquellas otras que sí lo están. A la primera parte de cada sumador completo la llamaremos *sumador completo parcial* (PFA, *partial full adder*). Esta separación se muestra en la Figura 5-6(a), que presenta el diagrama de un PFA y los diagramas de cuatro PFAs conectados al camino de propagación del acarreo. Hemos quitado la puerta OR y una de las puertas AND de cada uno de los sumadores completos para formar el camino del acarreo serie.

Cada PFA tiene dos salidas, P_i y G_i, hacia el camino del acarreo serie y una entrada C_i desde el camino al PFA. La función se llama *función de propagación*. Siempre que P_i sea igual a 1, un acarreo entrante se propaga a través de la posición del bit desde C_i a C_{i+1}. Para $P_i = 0$, la propagación de acarreo a través de la posición del bit se detiene. La función se llama *función generación*. Siempre que G_i sea igual a 1, la salida del acarreo de la posición es 1, sin tener en cuenta el valor de P_i, generándose entonces un acarreo en la posición. Cuando G_i es 0, no se genera acarreo, entonces C_{i+1} es 0 si el acarreo propagado a través de la posición C_i también es 0. Las funciones de generación y propagación corresponden exactamente a las del semi-sumador y son esenciales para controlar los valores del acarreo serie en el camino. Igual que el sumador completo, el PFA realiza la función de la suma mediante la OR exclusiva del acarreo entrante C_i y la función de propagación P_i.

El camino del acarreo en un sumador de 4 bits con acarreo serie está formado por un total de ocho puertas en cascada, por lo que el circuito tiene un retardo de ocho retardos de puerta. Puesto que el camino del acarreo está formado sólo por puertas OR y AND, idealmente el retardo para cada una de las cuatro señales de acarreo producidas, desde C_1 hasta C_4, podría ser de sólo dos retardos de puerta. El circuito básico para el acarreo paralelo es simplemente un circuito en el que las funciones de C_1 a C_3 tienen un retardo de sólo dos retardos de puerta. La implementación de C_4 es más complicada pues debe permitir que el sumador de 4 bits con acarreo anticipado pueda extenderse a múltiplos de 4 bits, por ejemplo a 16 bits El circuito para el acarreo paralelo de 4 bits se muestra en Figura 5-6(b). Está diseñado para reemplazar directamente el camino de acarreo serie de la Figura 5-6(a). Puesto que la generación lógica de C_1 ya está en dos niveles, permanece sin modificaciones. La lógica para C_2, sin embargo, tiene cuatro niveles. Para encontrar la lógica del acarreo paralelo para C_2, debemos reducir la lógica a dos niveles. La ecuación para C_2 se obtiene de la Figura 5-6(a), y se le aplica la ley distributiva para obtener:

$$\begin{aligned} C_2 &= G_1 + P_1(G_0 + P_0C_0) \\ &= G_1 + P_1G_0 + P_1P_0C_0 \end{aligned}$$

La salida C_2 de la Figura 5-6(b) es la implementación lógica de esta ecuación. Del mismo modo, a partir de la ecuación encontrada en el camino del acarreo de la Figura 5-6(a) y aplicando la ley distributiva, hemos obtenido la lógica en dos niveles para C_3:

$$\begin{aligned} C_3 &= G_2 + P_2(G_1 + P_1(G_0 + P_0C_0)) \\ &= G_2 + P_2(G_1 + P_1G_0 + P_0C_0) \\ &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \end{aligned}$$

Esta función se implementa mediante lógica a dos niveles con la salida C_3 de la Figura 5-6(b).

Podríamos implementar C_4 usando el mismo método. Pero algunas de las puertas tendrían un fan-in de cinco, lo que puede aumentar el retardo. Estamos interesados también en reutilizar este mismo circuito para bits superiores (por ejemplo, del 4 al 7, del 8 al 11, y del 12 al 15 en

un sumador de 16 bits). Para este sumador nos gustaría que el acarreo para las posiciones 4, 8 y 12, fuese generado lo más rápido posible y sin requerir un excesivo fan-in. Por tanto queremos repetir este truco del acarreo paralelo para grupos de 4 bits. Esto nos permitirá reutilizar el circuito de acarreo paralelo para cada grupo de 4 bits, y también usar el mismo circuito para cuatro grupos de 4 bits como si fueran bits individuales. Así en lugar de generar C_4, obtenemos funciones de generación y propagación que se aplican a grupos de 4 bits en lugar de aplicarse a un único bit y que actúan como entradas para el grupo de circuitos de acarreo paralelo. Para propagar un acarreo desde C_0 hasta C_4 necesitamos que las cuatro funciones de propagación sean iguales a 1, dando la función *propagación de grupo*

$$P_{0-3} = P_3P_2P_1P_0$$

Para representar la generación de un acarreo en las posiciones 0, 1, 2, y 3, y su propagación a C_4, necesitamos considerar la generación del acarreo en cada una de las posiciones, representado por G desde G_0 hasta G_3, y la propagación de cada uno de estos cuatro acarreos generados hasta la posición 4. Esto da la función *generación de grupo*

$$G_{0-3} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

Las ecuaciones para la propagación de grupo y para la generación de grupo se implementan mediante lógica en la parte inferior de la Figura 5-6(b). Si sólo hay 4 bits en el sumador, entonces el circuito lógico empleado para C_1 puede usarse para generar C_4 a partir de estas dos salidas. En un sumador más grande, se coloca un circuito de acarreo paralelo idéntico al de la figura, excepto por los nombres de las señales, en el segundo nivel para generar C_4, C_8 y C_{12}. Este concepto se puede ampliar a un sumador de 64 bits con más circuitos de acarreo paralelo en el segundo nivel y con un circuito de acarreo paralelo en el tercer nivel para generar los acarreos de las posiciones 16, 32 y 48.

Suponiendo que una OR exclusiva contribuye con 2 retardos de puerta, el mayor retardo que se produce en el sumador de 4 bits con acarreo anticipado es de 6 retardos de puerta, frente a los 10 retardos de puerta del sumador con acarreo serie. La mejora es muy modesta y quizás no compensa toda la lógica extra. Pero aplicando el circuito de acarreo paralelo a un sumador de 16 bits que emplea cinco copias en dos niveles reduce el retraso de 34 a sólo 10 retardos de puerta, lo que mejora la actuación del sumador por un factor de cerca de tres. En un sumador de 64 bits, empleando 21 circuitos de acarreo paralelo en tres niveles, el retardo se reduce de 130 a 14 retardos de puerta, dando un factor de más de 8 en la mejora de la actuación. En general, para la implementación que hemos mostrado, el retardo de un sumador con acarreo paralelo diseñado para optimizar el funcionamiento es de $4L + 2$ retardos de puerta, dónde L es el número de niveles del generación anticipada de acarreo del diseño.

5-3 RESTA BINARIA

En el Capítulo 1, examinamos brevemente la resta de números binarios sin signo. Aunque los textos introductorios sólo cubren la suma y la resta de números con signo, excluyendo completamente la alternativa sin signo, la aritmética de números sin signo juega un importante papel en la computación y en el diseño del hardware de computadoras. Se usa en unidades de punto flotante, en sumas con magnitud y signo y en algoritmos de resta, así como para extender la precisión de números de punto fijo. Por estas razones, trataremos aquí la suma y la resta de números sin signo. Sin embargo, también hemos escogido tratarlo primero para poder justificar clara-

mente, basándonos en el coste del hardware, el uso, a priori extraño y a menudo admitido como acto de fe, de las representaciones aritméticas en complemento.

En la Sección 1-3, la resta se realizó comparando el substraendo con el minuendo y restando el menor del mayor. El empleo de un método que recurre a esta operación de comparación produce una circuitería ineficaz y costosa. Como alternativa, podemos restar el substraendo simplemente del minuendo. Usando los mismos números que en el ejemplo de resta de la Sección 1-3, tenemos

Acarreos:	**1**1100
Minuendo	10011
Substraendo	−11110
Diferencia	10101
Diferencia correcta	−01011

Si no hay ningún acarreo en la posición más significativa, entonces sabemos que el substraendo es menor que el minuendo y, por tanto, el resultado es positivo y correcto. Si ocurre un acarreo en la posición más significativa, indicado en azul, entonces sabemos que el substraendo es mayor que el minuendo. Entonces, el resultado debe ser negativo, y necesitaremos corregir su magnitud. Podemos hacer esto examinando el resultado del cálculo siempre que ocurra un acarreo:

$$M - N + 2^n$$

Observe que el sumando 2^n representa el valor del acarreo en la posición más significativa. En lugar de este resultado, la magnitud deseada es $N - M$, que se obtiene restando la fórmula anterior de 2^n:

$$2^n - (M - N + 2^n) = N - M$$

En el ejemplo anterior, $100000 - 10101 = 01011$ que es la magnitud correcta.

En general, la resta de dos números de n dígitos, $M - N$, en base 2 puede hacerse como sigue:

1. Reste el substraendo N del minuendo M.
2. Si no hay ningún acarreo final, entonces $M \geqslant N$, y el resultado es positivo y correcto.
3. Si se produce un acarreo final, entonces $N > M$, y la diferencia, $M - N + 2^n$ se resta de 2^n y se añade un signo menos al resultado.

La resta de un número binario de 2^n para obtener un resultado de n dígitos se denomina *complemento a* 2 del número. Así en el paso 3, estamos tomando el complemento a 2 de la diferencia $M - N + 2^n$. El empleo del complemento a 2 en la resta se ilustra con el siguiente ejemplo.

EJEMPLO 5-1 Resta binaria sin signo en complemento a 2

Realice la resta binaria 01100100 − 10010110. Tenemos:

Acarreos:	**1**0011110
Minuendo:	01100100
Substraendo:	−10010110
Resultado inicial	11001110

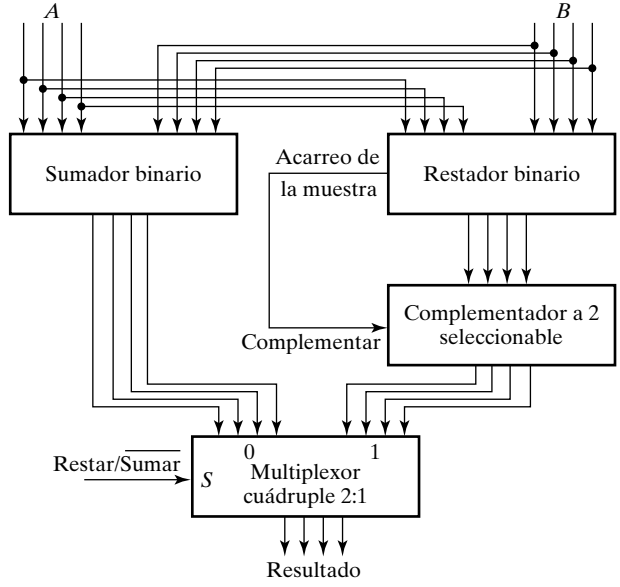
El acarreo final a 1 implica la corrección:

2^8	100000000
— Resultado inicial	−11001110
Resultado final	−00110010

■

Para implementar una resta empleando este método se necesita un restador para la resta inicial. Además, cuando sea necesario, se utilizará un segundo restador o bien un complementador a 2 para realizar la corrección. Hasta ahora, por tanto, necesitamos un restador, un sumador, y posiblemente un complementador a 2 para efectuar la suma y la resta. El diagrama de bloques del sumador-restador de 4 bits que emplea dichos bloques funcionales se muestra en la Figura 5-7. Las entradas se aplican tanto al sumador como al restador, de modo que ambos realicen las operaciones en paralelo. Si durante la resta se produce un acarreo de 1 en el extremo, entonces el complementador a 2 selectivo recibe en su entrada de complemento un 1. Este circuito saca entonces el complemento a 2 de las salidas del restador. Si el acarreo final tiene valor de 0, el complementador a 2 selectivo muestra las salidas del restador sin cambios. Si es una operación de resta, entonces se aplica un 1 a la entrada *S* del multiplexor seleccionando las salidas del complementador. Si es una operación de suma, entonces se aplica un 0 a *S*, seleccionando de este modo las salidas del sumador.

Tal y como veremos, este circuito es más complejo de lo necesario. Para reducir la cantidad de hardware, nos gustaría compartir la lógica entre el sumador y el restador. Esto también se puede hacer empleando la notación del complemento. Así, antes de ir más allá en las consideraciones sobre la combinación sumador-restador, estudiaremos con más detalles los complementos.



□ **FIGURA 5-7**
Diagrama de bloques de un sumador-restador binario

Complementos

Hay dos tipos de complementos para cada sistema en base-*r*: el complemento a la base, que ya vimos para base 2, y el complemento a la base menos 1. El primero se denomina *complemento a r* y el segundo se denomina *complemento a* $(r - 1)$. Cuando el valor de la base *r* se sustituye por los nombres, los dos tipos mencionados son el complemento a 2 y el complemento a 1 para los números binarios, y el complemento a 1, 0 y a 9 para los números decimales, respectivamente. Sin embargo, ahora nuestro interés está en los números binarios y en sus operaciones, y sólo trataremos de los complementos a 1 y a 2.

A partir de un número binario *N* de *n* dígitos, se define su *complemento a* 1 de *N* como $(2^n - 1) - N$. 2^n se representa por un número binario que consiste en un 1 seguido por tantos 0 como *n*. $2^n - 1$ es un número binario representado por *n* 1. Por ejemplo, si $n = 4$, tenemos $2^4 = (10000)_2$ y $2^4 - 1 = (1111)_2$. Así, el complemento a 1 de un número binario se obtiene restando cada dígito de 1. Al restar los dígitos binarios de 1, puede ocurrir $1 - 0 = 1$ ó $1 - 1 = 0$, lo que provoca el cambio del bit original de 0 a 1 o de 1 a 0, respectivamente. Por consiguiente, el complemento a 1 de un número binario se forma cambiando todos los 1 por 0 y todos los 0 por 1 —esto es, aplicando NOT o la función complemento a cada uno de los bits. Los siguientes son dos ejemplos numéricos:

El complemento a unos de 1011001 es 0100110.
El complemento a unos de 0001111 es 1110000.

De manera similar, el complemento a 9 de un número decimal, el complemento a 7 de un número octal, y el complemento a 15 de un número hexadecimal se obtiene restando cada dígito de 9, 7, y F (decimal 15), respectivamente.

Dado un número *N* de *n* dígitos en binario, el complemento a 2 de *N* se define como $2^n - N$ para $N \neq 0$ y 0 para $N = 0$. La razón para el caso especial de $N = 0$ es que el resultado debe tener *n* bits, y la resta de 0 de 2^n da un resultado de $(n + 1)$ bits. Este caso especial se consigue empleando un único restador de *n* bits o retirando el 1 de la posición extra. Comparándolo con el complemento a 1, observamos que el complemento a 2 se obtiene sumando un 1 al complemento a 1, es decir $2^n - N = \{[(2^n - 1) - N] + 1\}$. Por ejemplo, el complemento a 2 del binario 101100 es $010011 + 1 = 010100$ y se obtiene sumando 1 al valor del complemento a 1. De nuevo, para $N = 0$, el resultado de esta suma es 0, y se ha conseguido ignorando el acarreo externo de la posición más significativa de la suma. Estos conceptos también se mantienen para otras bases. Como veremos más tarde, son muy útiles para simplificar el complemento a 2 y el hardware de la resta.

El complemento a 2 también se puede formar dejando inalterados todos los 0 menos significativos y el primer 1, y reemplazando entonces los 1 por 0 y los 0 por 1 en los restantes bits más significativos. Así, el complemento a 2 de 1101100 es 0010100 y se obtiene dejando igual los dos 0 de menor orden y el primer 1 y reemplazando en los otros 4 bits más significativo los 1 por 0 y los 0 por 1. En otras bases, el primer dígito no nulo se resta de la base *r*, y se reemplazan los dígitos restantes a la izquierda por dígitos de valores $r - 1$.

También es importante mencionar que el complemento del complemento devuelve el número a su valor original. Para ver esto, observe que el complemento a 2 de *N* es $2^n - N$, y el complemento del complemento es $2^n - (2^n - N) = N$, devolviendo el número original.

Resta con complementos

Con anterioridad, comentamos nuestro deseo de simplificar el hardware compartiendo la lógica del sumador y del restador. Respaldados por los complementos nos preparamos a definir un pro-

cedimiento para la resta binaria que emplea la suma y la lógica del complemento correspondiente. La resta binaria de dos números de n dígitos sin signo $M - N$, puede realizarse del siguiente modo:

1. Sume el complemento a 2 del substraendo N al minuendo M.

$$M + (2^n - N) = M - N + 2^n$$

2. Si $M \geqslant N$, la suma producirá un acarreo final, 2^n. Deseche el acarreo final, dejando mientras como resultado $M - N$.
3. Si $M < N$, la suma no producirá acarreo final por lo que será igual a $2^n - (N - M)$, el complemento a 2 de $N - M$. Realice una corrección tomando el complemento a 2 de la suma y poniendo un signo menos delante para obtener el resultado $-(N - M)$.

Los ejemplos que aparecen más adelante ilustran el procedimiento anterior. Note que, aunque estamos tratando con números sin signo, no hay ninguna manera de conseguir un resultado sin signo para el caso del paso 3. Al trabajar con papel y lápiz, reconocemos, por la ausencia del acarreo final, cuándo la respuesta debe cambiarse a un número negativo. Si queremos conservar el signo menos del resultado, se debe guardar separadamente del resultado corregido de n bits.

EJEMPLO 5-2 Resta binaria sin signo en complemento a 2

Dados dos números binarios $X = 1010100$ e $Y = 1000011$, realice la resta $X - Y$ e $Y - X$ usando operaciones en complemento a 2. Tenemos:

$$\begin{aligned} X &= 1010100 \\ \text{Complemento a 2 de } X &= 0111101 \\ \text{Suma} &= 10010001 \\ \text{Ignorar el acarreo } 2^7 &= -\underline{10000000} \\ \text{Respuesta: } X - Y &= 0010001 \\ Y &= 1000011 \\ \text{Complemento a 2 de } X &= \underline{0101100} \\ \text{Suma} &= 1101111 \end{aligned}$$

No hay acarreo final.

Respuesta: $Y - X = -(\text{complemento a 2 de } 1101111) = -0010001$ ■

La resta de números sin signo también se puede hacer mediante el complemento a 1. Recuerde que el complemento a 1 es 1 menos que el complemento a 2. Debido a esto, el resultado de sumar el minuendo al complemento del substraendo produce una suma que es 1 menos que de la diferencia correcta cuando se produce acarreo final. Desechar el acarreo final y sumar 1 a la suma se denomina *redondeo del acarreo*.

EJEMPLO 5-3 Resta binaria sin signo mediante sumas en complemento a 1

Repita el Ejemplo 5-2 usando operaciones en complemento a 1. Aquí, tenemos:

$$
\begin{aligned}
X - Y &= 1010100 - 1000011 \\
X &= \quad 1010100 \\
\text{Complemento a uno de } Y &= + \underline{\ 0111100} \\
\text{Suma} &= \quad \mathbf{1}0010000 \\
\text{Redondeo del acarreo} &= \quad \underline{\longrightarrow +\mathbf{1}} \\
\textit{Respuesta: } X - Y &= \quad 0010001 \\
Y - X &= 1000011 - 1010100 \\
Y &= \quad 1000011 \\
\text{Complemento a uno de } X &= + 0101011 \\
\text{Suma} &= \quad \underline{1101110} \\
&\text{No hay acarreo final.}
\end{aligned}
$$

Respuesta: $Y - X = -$(complemento a uno de 1101110) $= -0010001$ ■

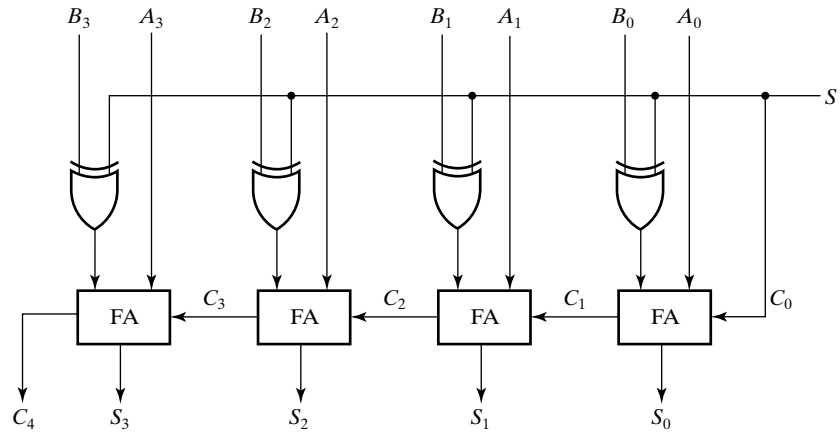
Observe que el resultado negativo se obtiene tomando el complemento a 1 de la suma, ya que éste es el tipo de complemento empleado.

5-4 SUMADOR-RESTADOR BINARIO

Usando el complemento a 1 y el complemento a 2 hemos suprimido la operación de la resta y solamente necesitamos un complementador apropiado y un sumador. Cuando realizamos una resta complementamos el substraendo *N*, mientras que para sumar no es necesario. Estas operaciones se pueden lograr usando un complementador selectivo y un sumador interconectados para formar un sumador-restador. Empleamos el complemento a 2 por ser el más habitual en los sistemas modernos. El complemento a 2 se puede obtener tomando el complemento a 1 y sumándole 1 al bit de menor peso. El complemento a 1 se implementa fácilmente a partir de circuitos inversores, mientras que la suma de 1 la conseguimos haciendo la entrada de acarreo del sumador paralelo igual a 1. De esta manera, usando el complemento a 1 y una entrada no usada del sumador, se obtiene muy económicamente el complemento a 2. En la resta en complemento a 2, como en el paso de corrección después de la suma, tenemos que complementar el resultado y añadirle un signo menos si se produce acarreo final. La operación de corrección se realiza empleando cualquier sumador-restador una segunda vez con $M = 0$ o un complementador selectivo como el de la Figura 5-7.

El circuito para restar $A - B$ consiste en un sumador paralelo como el de la Figura 5-5, con inversores colocados entre cada términal B y la entrada correspondiente del sumador completo. La entrada de acarreo C_0 debe ser igual a 1. La operación que se realiza se convierte en: $A +$ complemento a 1 de $B + 1$. Esto es igual a $A +$ el complemento a 2 de B. Para los números sin signo, da $A - B$ si $A \geqslant$ o el complemento a 2 de $B - A$ si $A < B$.

La suma y la resta se pueden combinar en un circuito con un sumador binario común. Esto se hace incluyendo una puerta OR exclusiva con cada sumador completo. En la Figura 5-8 se muestra un circuito sumador-restador de 4 bits. La entrada S controla el funcionamiento. Cuando $S = 0$ el circuito funciona como un sumador, y cuando $S = 1$ el circuito se convierte en un restador. Cada puerta OR exclusiva recibe la entrada S y una de las entradas de B, B_i. Cuando $S = 0$, tenemos $B_i \oplus 0$. Si los sumadores completos reciben el valor de B, y la entrada de acarreo es 0, el circuito realiza $A + B$. Cuando $S = 1$, tenemos $B_i \oplus 1 = \overline{B_i}$ y $C_0 = 1$. En este caso, el circuito realiza la operación $A +$ el complemento a 2 de B.



□ **FIGURA 5-8**
Circuito sumador/restador

Números binarios con signo

En la sección anterior tratamos la suma y la resta de números sin signo. Ahora extenderemos esta aproximación a los números con signo, incluyendo otros usos de los complementos que nos ayudarán a eliminar el paso de la corrección.

Los enteros positivos y el número 0 pueden representarse como números sin signo. Para representar los enteros negativos, necesitamos, sin embargo, una notación específica para los valores negativos. En aritmética ordinaria, un número negativo se indica por un signo menos y un número positivo por un signo más. Debido a las limitaciones del hardware, las computadoras deben representar todo con 1 y 0, incluso el signo de un número. En consecuencia, es costumbre representar el signo con un bit colocado en la posición más significativa del número de *n* bits. Lo convencional es emplear el 0 para los números positivos y el 1 para los números negativos.

Es importante comprender que ambos, los números binarios sin signo y los números binarios con signo, consisten en una cadena de bits cuando se representan en una computadora. El usuario es quien determina si el número es con signo o sin signo. Si el número binario es con signo, entonces el bit del extremo izquierdo representa el signo y el resto de los bits representan el número. Si se supone que el número binario es sin signo, entonces el bit del extremo izquierdo es el bit de mayor peso del número. Por ejemplo, la cadena de bits 01001 puede ser considerada como el 9 (en binario sin signo) o el $+9$ (en binario con signo), puesto que el bit del extremo izquierdo es 0. De manera similar, la cadena de bits 11001 representa el equivalente binario de 25 cuando se considera como un número sin signo ó de -9 cuando se considera como un número con signo. Esto último es porque el 1 en la posición del extremo izquierdo designa un signo menos y los restantes cuatro bits representan el 9 binario. Normalmente, no hay confusión identificando los bits ya que se conoce de antemano el tipo de representación del número. Esta representación de números con signo se denomina sistema de representación en *signo y magnitud*. En este sistema, el número consiste en una magnitud y un símbolo ($+$ ó $-$) o un bit (0 ó 1) indicando el signo. Ésta es la representación de números con signo que se emplea en la aritmética ordinaria.

En la implementación de sumas y restas para números de *n* bits en *signo y magnitud*, el bit de signo —en el extremo izquierdo— y los $n - 1$ bits de magnitud se procesan separadamente. Los bits de magnitud se procesan como en los números binarios sin signo. Por tanto, la resta

implica necesariamente tener que llevar a cabo el paso de la corrección. Para evitar este paso empleamos un sistema de representación diferente para los números negativos, llamado sistema en *signo y complemento*. En este sistema un número negativo se representa por su complemento. Mientras en el sistema de signo y magnitud un número se convierte en negativo cambiando su signo, en el sistema de signo y complemento se hace tomando su complemento. Puesto que los números positivos empiezan siempre con un 0 (representando un signo +) en la posición del extremo izquierdo, sus complementos empezarán siempre con un 1, indicando un número negativo. Los sistemas de signo y complemento pueden usar tanto el complemento a 1 como a 2, pero éste último es el más común. Como ejemplo, considere el número 9, representado en binario con 8 bits. Se representa +9 con un bit de signo a 0 en el extremo izquierdo, seguido por el equivalente binario de 9, dando 00001001. Observe que los 8 bits deben tener un valor, y por consiguiente, se insertan 0 entre el bit de signo y el primer 1. Aunque sólo hay una manera de representar +9, tenemos tres maneras diferentes de representar −9 usando 8 bits:

En la representación de signo y magnitud:	10001001
En la representación de signo y complemento a 1:	11110110
En la representación de signo y complemento a 2:	11110111

En signo y magnitud, se obtiene −9 de +9 cambiando el bit de signo del extremo izquierdo de 0 a 1. En signo y complemento a 1, −9 se obtiene complementando todos los bits de +9, incluso el bit de signo. La representación en signo y complemento a 2 de −9 se obtiene tomando el complemento a 2 del número positivo, incluso el bit 0 del signo.

La Tabla 5-3 lista las tres representaciones para todos los posibles números binarios de 4 bits con signo. También se muestra el número decimal equivalente. Observe cómo los números positivos son idénticos en todas las representaciones y tienen 0 en la posición del extremo izquierdo. En el sistema de signo y complemento a 2, el 0 tiene sólo una representación que es siempre positiva. Los otros dos sistemas tienen un 0 positivo y un 0 negativo, algo no habitual en la aritmética ordinaria. Observe también que todos los números negativos tienen un 1 en el bit del extremo izquierdo; ésta es la manera de distinguirlos de los números positivos. Con 4 bits, podemos representar 16 números binarios. En las representaciones en signo y magnitud y en complemento a 1, hay siete números positivos, siete números negativos y dos ceros con signo. En la representación en complemento a 2 hay siete números positivos un cero y ocho números negativos.

El sistema de signo y magnitud se emplea en la aritmética ordinaria, pero es incómodo en la aritmética de computación debido al manejo separado del signo y a la necesidad de emplear el paso de la corrección en la resta. Por consiguiente se utiliza normalmente el sistema de signo y complemento. El complemento a 1 supone ciertas dificultades debido a sus dos representaciones para el 0 y raramente se usa para las operaciones aritméticas. Es útil como operación, pues cambiar de 1 a 0 ó de 0 a 1 es equivalente a una operación lógica de complemento. En la siguiente discusión de aritmética binaria con signo se emplea exclusivamente la representación de números negativos en complemento a 2, pues es la que predomina en realidad. Para el complemento a 1 y el redondeo de acarreo se pueden aplicar los mismos procedimientos de signo que los que se han aplicado al complemento a 2 con signo.

Suma y resta binaria con signo

La suma de dos números, $M + N$, en el sistema de signo y magnitud sigue las reglas de la aritmética ordinaria: Si los dos signos son iguales, sumamos las magnitudes y damos el signo de M a la suma. Si los signos son diferentes, restamos la magnitud de N a la magnitud de M. La

□ **TABLA 5-3**
Números binarios con signo

Decimal	Signo y complemento a 2	Signo y complemento a unos	Signo y magnitud
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
−0	—	1111	1000
−1	1111	1110	1001
−2	1110	1101	1010
−3	1101	1100	1011
−4	1100	1011	1100
−5	1011	1010	1101
−6	1010	1001	1110
−7	1001	1000	1111
−8	1000	—	—

ausencia o presencia de acarreo final determina el signo del resultado, basado en el signo de *M*, y determina si hay o no que realizar una corrección del complemento a 2. Por ejemplo, puesto que los signos son diferentes, (0 0011001) + (1 0100101) provoca que 0100101 sea restado de 0011001. El resultado es 1110100, y se ha producido un acarreo de 1. El acarreo final indica que la magnitud de *M* es menor que la magnitud de *N*. Entonces, el signo del resultado es opuesto al de *M* y es, por consiguiente, un menos (−). El acarreo final también indica que la magnitud del resultado, 1110100, se debe corregir tomando su complemento a 2. Combinando el signo y la magnitud corregida del resultado, se obtiene 1 0001100.

A diferencia del sistema de signo y magnitud, la regla para sumar números en signo y complemento no requiere ni comparación ni resta, sino sólo suma. El procedimiento es simple y puede exponerse como sigue para los números binarios:

> La suma de dos números binarios con signo, con números negativos representados en signo y complemento a 2, se obtiene sumando los dos números, incluyendo sus bits de signo y desechando cualquier acarreo que se produzca más allá del bit de signo.

En el Ejemplo 5-4 se dan algunos ejemplos numéricos de suma binaria con signo. Observe que los números negativos ya están en complemento a 2 y que el resultado obtenido tras la suma, si es negativo, queda representado de la misma forma..

EJEMPLO 5-4 Suma binaria con signo empleando complemento a 2

+6	00000110	−6	11111010	+6	00000110	−6	11111010
+13	00001101	+13	00001101	−13	11110011	−13	11110011
+19	00010011	+7	00000111	−7	11111001	−19	11101101

En cualquiera de los cuatro casos, la operación realizada ha sido la suma, incluyendo el bit de signo. El acarreo que se produzca más allá del bit de signo se desecha, y los resultados negativos están automáticamente en forma de complemento a 2. ■

La manera de representar números negativos en complemento es poco familiar para quienes están acostumbrados al sistema de signo y magnitud. Para determinar el valor de un número negativo en la forma de signo y complemento a 2 se necesita convertir el número en un número positivo, de manera que nos sea más familiar. Por ejemplo, el número binario con signo 11111001 es negativo porque el bit del extremo izquierdo es 1. Su complemento a 2 es 00000111 que equivale al binario $+7$. De este modo reconocemos el número original como -7.

La resta de dos números binarios con signo, cuando los números negativos están en forma de complemento a 2, es muy simple y se puede exponer como sigue:

> Tome el complemento a 2 del substraendo (incluso el bit de signo) y súmelo al minuendo (incluso el bit de signo). No se considerará ningún acarreo más allá del bit de signo.

Este procedimiento viene del hecho de que una operación de resta puede cambiarse por una operación de suma si el signo del substraendo se cambia. Es decir,

$$(\pm A) - (+B) = (\pm A) + (-B)$$
$$(\pm A) - (-B) = (\pm A) + (+B)$$

Cambiar un número positivo por un número negativo es fácil si se toma su complemento a 2. Al contrario también es verdad, porque el complemento de un número negativo, que ya está en forma de complemento, produce el número positivo correspondiente. Se muestran algunos ejemplos numéricos en el Ejemplo 5-5.

EJEMPLO 5-5 Resta binaria con signo que usa el complemento a 2

-6	11111010	11111010	$+6$	00000110	00000110
$-(-13)$	-11110011	$+00001101$	$-(-13)$	-11110011	$+00001101$
$+7$		00000111	$+19$		00010011

El acarreo final se desecha. ■

Es importante mencionar que los números binarios en el sistema de signo y complemento se suman y restan con las mismas reglas básicas que se aplican para los números sin signo. Por consiguiente, las computadoras necesitan un único circuito hardware común para ocuparse de ambos tipos de aritmética. El usuario o el programador deben interpretar de manera diferente los resultados de tales sumas o restas, en tanto que depende de si se han supuesto los números con signo o sin signo. De este modo, el mismo sumador-restador diseñado para números sin signo se puede emplear para números con signo.

Si los números con signo están representados en complemento a 2 entonces el circuito de la Figura 5-8 se puede usar sin el paso de la corrección. Para sumas y restas en complemento a 1, la entrada desde S a C_0 del sumador se reemplaza por una entrada desde C_n a C_0.

Overflow o desbordamiento

Para obtener un resultado correcto cuando se suma y se resta, debemos asegurarnos que el resultado tenga el número suficiente de bits para acomodar la suma. Decimos que se ha producido

overflow o desbordamiento, si partiendo de dos números de n bits su suma ocupa $n + 1$ bits. Esto es cierto para números binarios o números decimales con o sin signo. Cuando se realiza la suma con papel y lápiz el desbordamiento no es un problema, ya que no estamos limitados por la anchura de la página. Nos basta con añadir en la posición más significativa otro 0 a un número positivo y otro 1 a un número negativo para extenderlos a $n + 1$ bits y entonces poder realizar la suma. El desbordamiento es un problema en las computadoras porque el número de bits que soporta un número es fijo, y no puede acomodarse un resultado que excede del número de bits. Por esta razón, las computadoras detectan y pueden señalar la aparición del overflow. La detección del desbordamiento puede manejarse automáticamente interrumpiendo la ejecución del programa y tomando una acción especial. Una alternativa es monitorizar o supervisar mediante software las condiciones del desbordamiento.

Detectar la aparición del desbordamiento tras la suma de dos números binarios dependerá de si hemos considerado los números con o sin signo. Cuando se suman dos números sin signo el desbordamiento se detecta si ha habido acarreo más allá de la posición más significativa. En la resta sin signo, la magnitud del resultado es siempre igual a o menor que el mayor de los números originales, con lo que el desbordamiento es imposible. En el caso de los números con signo en complemento a 2, el bit más significativo siempre representa el signo. Cuando se suman dos números con signo, el bit de signo se trata como una parte del número y un acarreo de 1 en el extremo no implica necesariamente un desbordamiento.

En la suma de números con signo el desbordamiento nunca se produce si un número es positivo y el otro es negativo: sumar un número positivo a un número negativo produce un resultado cuya magnitud es igual o menor que el mayor de los números originales. El desbordamiento sólo ocurre si los dos números sumados son ambos positivos o negativos. Para ver cómo se produce esto considere el siguiente ejemplo en complemento a 2: se almacenan dos números con signo $+70$ y $+80$ en dos registros de 8 bits. El rango de números binarios que cada registro puede almacenar, expresado en decimal, es de $+127$ a -128. Por tanto, la suma de los dos números almacenados, $+150$, excede de la capacidad del registro de 8 bits. También ocurre así para -70 y -80. Estas dos sumas, junto con los valores de los bits de acarreo más significativos, son las siguientes:

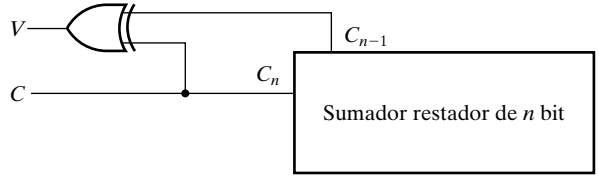
Acarreos:	0 1	Acarreos:	1 0
$+70$	0 1000110	-70	1 0111010
$+80$	0 1010000	-80	1 0110000
$+150$	1 0010110	-150	0 1101010

Observe que el resultado de 8 bits que debía haber sido positivo tiene un bit de signo negativo y que el resultado de 8 bits que debía ser negativo tiene un bit de signo positivo. Sin embargo, si consideramos el acarreo producido más allá del bit de signo como el signo del resultado, entonces la respuesta obtenida con 9 bits es correcta. Puesto que, a pesar de todo, no hay lugar para el noveno bit del resultado, decimos que ha ocurrido un desbordamiento.

La condición de desbordamiento se puede detectar observando el acarreo asociado a la posición del bit de signo el acarreo anterior. Si estos dos acarreos no son iguales, hay desbordamiento. Esto se aprecia sencillamente en el ejemplo que hemos realizado en complemento a 2, dónde los dos acarreos se muestran explícitamente. Hay desbordamiento si al aplicar los dos acarreos a una puerta OR exclusiva, la salida de la puerta es igual a 1. Para trabajar correctamente en complemento a 2, hay que aplicar el complemento a 1 del substraendo al sumador y sumarle 1 ó bien dotar de deteccion de desbordamiento al circuito complementador a 2. La segunda opción es debida a la posibilidad de que se produzca desbordamiento cuando el substraendo es el menor número negativo.

La Figura 5-9 muestra la sencilla lógica necesaria para detectar la aparición del desbordamiento. Si se consideran los números sin signo, entonces la salida *C* detecta un acarreo (un desbordamiento) para la suma cuando es igual a 1 y para el caso de la resta indica que no se necesita ningún paso de corrección. Si *C* es igual a 0 no detecta acarreo (ningún desbordamiento) para la suma mientras que en el caso de la resta indica que se necesita corregir el resultado.

Si los números son con signo, entonces se emplea la salida *V* para detectar un desbordamiento. Si $V = 0$ después de una suma con signo o una resta, indica que no hay desbordamiento y que el resultado es correcto. Si $V = 1$, entonces el resultado de la operación contiene $n + 1$ bits, pero sólo los *n* bits de la derecha encajan en el resultado de *n* bits, y por tanto, se ha producido desbordamiento. El bit $n+1$ es el signo real del resultado, aunque no puede ocupar la posición del bit de signo en el resultado.



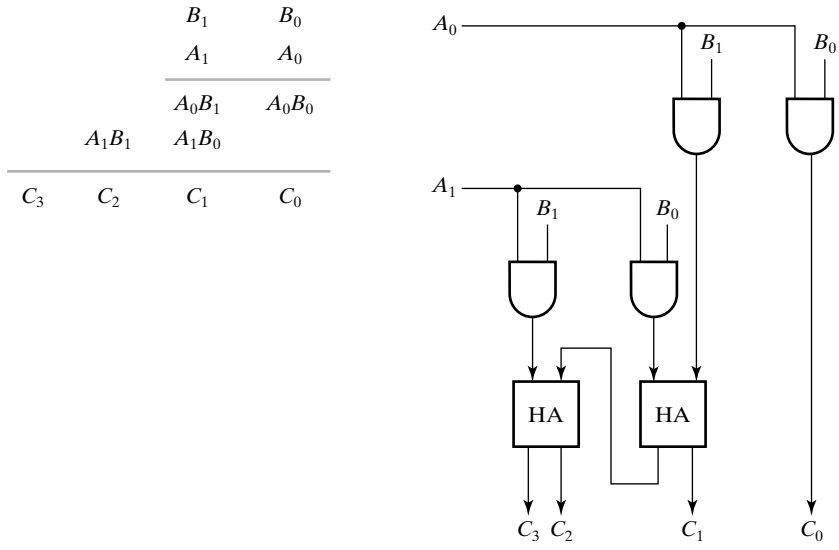
□ **FIGURA 5-9**
Lógica para la detección del desbordamiento en una suma o resta

5–5 MULTIPLICACIÓN BINARIA

La multiplicación de números binarios se realiza de la misma manera que con los números decimales. El multiplicando se multiplica por cada bit del multiplicador, empezando por el bit menos significativo. Cada una de estas multiplicaciones forma un producto parcial. Cada uno de estos productos parciales se desplaza un bit a la izquierda. El producto final se obtiene de la suma de los productos parciales.

Para ver cómo se puede implementar un multiplicador binario con un circuito combinacional, considere la multiplicación de dos números de 2 bits que se muestra en la Figura 5-10. Los bits del multiplicando son B_1 y B_0, los bits del multiplicador son A_1 y A_0, y el producto es C_3, C_2, C_1 y C_0. El primer producto parcial se forma multiplicando B_1 B_0 por A_0. La multiplicación de dos bits como A_0 y B_0 produce un 1 si ambos bits son 1; de lo contrario da como resultado un 0. Resulta idéntico al funcionamiento de una puerta AND. Por consiguiente, el producto parcial puede implementarse con puertas AND tal y como se ilustra en el diagrama. El segundo producto parcial se obtiene multiplicando B_1 B_0 por A_1 y desplazándolo una posición a la izquierda. Los dos productos parciales se suman con circuitos semi-sumadores (HA). Normalmente hay más bits en los productos parciales, por lo que será necesario usar sumadores completos para efectuar la suma de los productos parciales. Observe como, una vez obtenido a partir de la primera puerta AND, el bit menos significativo no tiene que pasar por el sumador.

El circuito combinacional de un multiplicador binario con más bits puede construirse de manera similar. Se realiza la AND de un bit del multiplicador con cada bit del multiplicando en tantos niveles como bits tenga el multiplicador. En cada nivel de puertas AND, la salida binaria se suma en paralelo con el producto parcial del nivel anterior para formar un nuevo producto parcial. El último nivel produce el producto final. Para un multiplicador de *J* bits y un multiplicando de *K* bits, necesitamos $J \times K$ puertas AND y $(J - 1)$ sumadores de *K* bits para producir un producto de $J + K$ bits. Como ejemplo de circuito combinacional de un multiplicador bina-



□ **FIGURA 5-10**
Multiplicador binario de 2 × 2 bits

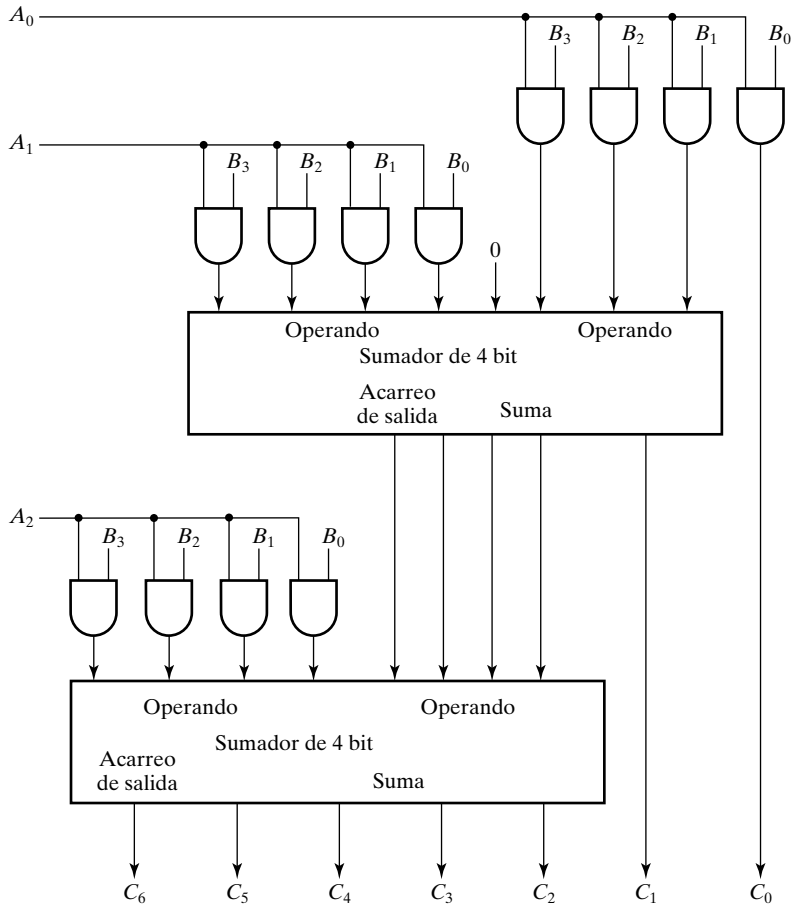
rio, considere un circuito que multiplica un número binario de cuatro bits por un número de tres bits. Representemos el multiplicando por $B_3B_2B_1B_0$ y el multiplicador por $A_2A_1A_0$. Puesto que $K = 4$ y $J = 3$, necesitamos 12 puertas AND y dos sumadores de 4 bits para obtener un producto de 7 bits. El diagrama lógico de este tipo de circuitos multiplicadores se muestra en la Figura 5-11. Observe que el bit de salida del acarreo forma parte del multiplicador entrando al sumador en el siguiente nivel.

5-6 OTRAS FUNCIONES ARITMÉTICAS

Hay otras operaciones aritméticas aparte de $+$, $-$, y $\times$ que son bastante importantes. Entre estas operaciones está el incremento, el decremento, la multiplicación y división por una constante y la comparación mayor que y menor que. Cada uno de estas operaciones puede implementarse para operandos de varios bits usando un array iterativo de células de 1 bit. En lugar de utilizar estas aproximaciones básicas, se emplean combinaciones de operaciones fundamentales y una nueva técnica denominada contracción. La contracción parte de un circuito como un sumador binario, un sumador con acarreo anticipado, o un multiplicador binario. Este enfoque simplifica el diseño convirtiendo los circuitos existentes en circuitos útiles, menos complicados en lugar de diseñar los últimos circuitos directamente.

Contracción o reducción

La asignación de valores, la transferencia o la inversión de las entradas se puede combinar con bloques funcionales para implementar nuevas operaciones tal y como se hizo en el Capítulo 4. Implementaremos nuevas operaciones usando técnicas similares sobre un circuito dado o sobre sus ecuaciones y reduciéndolo para una aplicación específica a un circuito más simple. Llamaremos a este procedimiento *contracción*. La meta de la contracción es lograr el diseño de un



□ **FIGURA 5-11**
Multiplicador binario de 4 bits × 3 bits

circuito lógico o de un bloque funcional usando los resultados de diseños previos. Esta técnica de contracción puede ser aplicada por el diseñador, para diseñar un circuito concreto, o por las herramientas de síntesis lógicas para simplificar un circuito inicial mediante la asignación de valores, la transferencia o la inversión en sus entradas. En ambos casos, la contracción también se puede aplicar sobre las salidas del circuito que no se emplean, contrayendo el circuito original hacia el circuito deseado. Ilustraremos primero la contracción usando ecuaciones booleanas.

EJEMPLO 5-6 Contracción de ecuaciones del sumador completo

El circuito a diseñar Add1 debe generar la suma S_i y el acarreo C_{i+1}, para la suma de un sólo bit $A_i + 1 + C_i$. Se trata del caso especial de un sumador completo $A_i + B_i + C_i$ con $B_i = 1$. Entonces, las ecuaciones para el nuevo circuito se pueden obtener a partir de las ecuaciones del sumador completo,

$$S_i = A_i \oplus B_i \oplus C_i$$

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$$

haciendo $B_i = 1$ y simplificando los resultados, obtenemos,

$$S_i = A_i \oplus 1 \oplus C_i = \overline{A_i \oplus C_i}$$

$$C_{i+1} = A_i \cdot 1 + A_i C_i + 1 \cdot C_i = A_i + C_i$$

Suponga que se emplea el circuito Add1 por cada uno de los cuatro sumadores completos en un sumador de 4 bits con acarreo serie. El cálculo a realizar será $S = A + 1111 + C_0$ en lugar de $S = A + B + C_0$. En complemento a 2, este cálculo es $S = A - 1 + C_0$. Si $C_0 = 0$, esto implementa una operación de decremento $S = A - 1$, empleando considerablemente menos lógica que la necesaria para un sumador-restador de 4 bits. ■

La contracción se puede aplicar a las ecuaciones, como se ha hecho aquí, o directamente sobre los diagramas del circuito aplicando funciones fundamentales a las entradas de los bloques funcionales. Para aplicar la contracción con éxito, la función deseada se debe poder obtener a partir del circuito inicial mediante la aplicación de funciones fundamentales en sus entradas. Ahora consideramos la contracción basada en las salidas no utilizadas.

Colocar un valor desconocido, X, en la salida de un circuito significa que dicha salida no se usará. De este modo, puede eliminarse tanto la salida de la puerta como cualquier otra puerta que esté conectada únicamente a dicha salida. Las reglas para acortar las ecuaciones con X en una o más salidas son las siguientes:

1. Eliminar todas las ecuaciones con X en las salidas del circuito.
2. Si alguna variable intermedia no aparece en ninguna de las restantes ecuaciones, eliminar su ecuación.
3. Si alguna variable de entrada no aparece en ninguna ecuación restante, eliminarla.
4. Repetir los pasos 2 y 3 hasta que no haya ninguna nueva anulación.

Las reglas para acortar un diagrama lógico con X en una o más de las salidas son las siguientes:

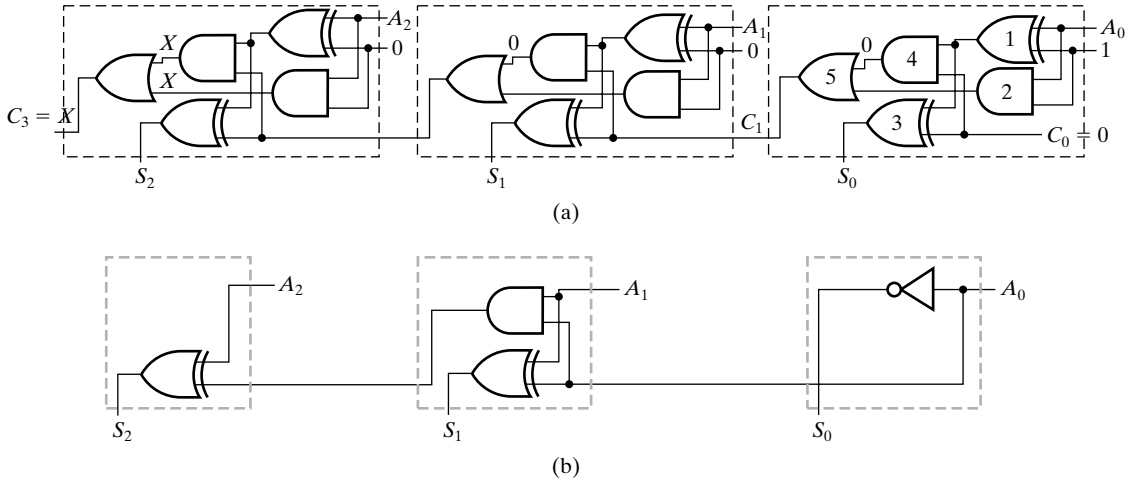
1. Empezando por las salidas, eliminar todas las puertas con X en sus salidas y colocar X en sus cables de entrada.
2. Si todos los cables atacados por una puerta están etiquetados con X, eliminar la puerta y colocar X en sus entradas.
3. Si todos los cables de entrada conectados a una entrada externa están etiquetados con X, elimine la entrada.
4. Repita 2 y 3 hasta que no sea posible ninguna nueva eliminación.

En la próxima subsección, se ilustra la contracción de un diagrama lógico para la operación de incremento.

Incremento

Incrementar significa añadir a una variable aritmética un valor fijo, el valor fijo empleado más a menudo es el 1. Un incrementador de n bits que realiza la operación $A + 1$ se puede obtener a partir de un sumador binario que realice la operación $A + B$ con $B = 0...01$. El uso de $n = 3$ es suficientemente grande para determinar la lógica necesaria para construir el circuito de un incrementador de n bits.

La Figura 5-12(a) muestra un sumador de 3 bits con las entradas conectadas de tal modo que representan el cálculo $A + 1$ y con la salida del bit de acarreo más significativo C_3 conectada al valor X. El operando $B = 001$ y el acarreo entrante $C_0 = 0$ hacen que el cálculo sea $A + 001 + 0$. Otra alternativa sería, $B = 000$ con el acarreo entrante $C_0 = 1$.



□ **FIGURA 5-12**
Contracción de un sumador a un incrementador

Basándonos en la asignación de valores, hay tres contracciones distintas para las células del sumador:

1. La célula menos significativa a la derecha con $B_0 = 1$ y $C_0 = 0$.
2. La célula normal en el medio con $B_1 = 0$.
3. La célula más significativa a la izquierda con $B_2 = 0$ y $C_3 = \mathsf{X}$.

Para la célula de la derecha, la salida de la puerta 1 es $\bar{A}_0$ por lo que puede reemplazarse por un inversor. La salida de la puerta 2 se vuelve A_0, por lo que puede reemplazarse por un cable conectado a A_0. Aplicando $\bar{A}_0$ y 0 a la puerta 3 ésta puede reemplazarse por un cable, que conecte A_0 con la salida S_0. La salida de la puerta 4 es 0, por lo que puede reemplazarse por un valor 0. Aplicando este 0 y A_0 desde la puerta 2 hasta la puerta 5, la puerta 5 puede reemplazarse por un cable que conecte A_0 con C_1. El circuito resultante es la célula de la derecha en la Figura 5-12(a).

Aplicando la misma técnica a la célula normal del centro con $B_1 = 0$ las salidas son:

$$S_1 = A_1 \oplus C_1$$
$$C_2 = A_1 C_1$$

El circuito resultante es la célula del centro en la Figura 5-12(b). Para la célula de la izquierda con $B_2 = 0$ y $C_3 = \mathsf{X}$, primero se propagan los efectos de X para ahorrar esfuerzos. Puesto que la puerta *A* tiene X en su salida, la salida se elimina y se ponen X en sus dos entradas. Como todas las puertas conectadas a las puertas *B* y *C* tienen X en sus entradas, pueden eliminarse y poner X en sus entradas. No pueden quitarse las puertas *D* y *E*, puesto que cada una está conectada a una puerta sin una X en su entrada. El circuito resultante se muestra como la célula izquierda en Figura 5-12(b).

Para un incrementador con $n > 3$ bits, la célula menos significativa del incrementador se coloca en la posición 0, la célula normal en posiciones de la 1 a la $n - 2$, y la célula más significativa en la posición $n - 1$. En este ejemplo, la célula del extremo derecho en la posición 1 es la contraída, pero, si desea, podría reemplazarse por la célula de la posición 2 con $B_0 = 0$ y $C_0 = 1$. Igualmente, podría generarse la salida C_3, pero no se usa. En ambos casos, se sacrifican el coste y la eficacia para hacer todas las células idénticas.

Decremento

Decrementar es sumar un valor negativo fijo a una variable aritmética, el valor fijo empleado más frecuentemente es -1. En el Ejemplo 5-6 ya se ha diseñado un decrementador. Otra posible alternativa sería diseñar un decrementador basándonos en un sumador-restador como circuito de partida, aplicando $B = 0...01$ y $C_0 = 0$, y seleccionando la operación de resta haciendo $S = 1$. Partiendo de un sumador-restador, también podemos usar la contracción para diseñar un circuito que incremente para $S = 0$ y decremente para $S = 1$ aplicando $B = 0...01$, $C_0 = 0$, y permitiendo a S seguir siendo una variable. En este caso, el resultado es una célula de la complejidad de un sumador completo en las posiciones de bits normales. De hecho, volviendo a los orígenes, redefiniendo la función de acarreo y diseñando aquella célula empleando esta redefinición, el coste pueden disminuirse un poco. De lo que se deduce que la contracción, aunque produce una implementación, puede no proporcionar el resultado con menor coste o mejor rendimiento.

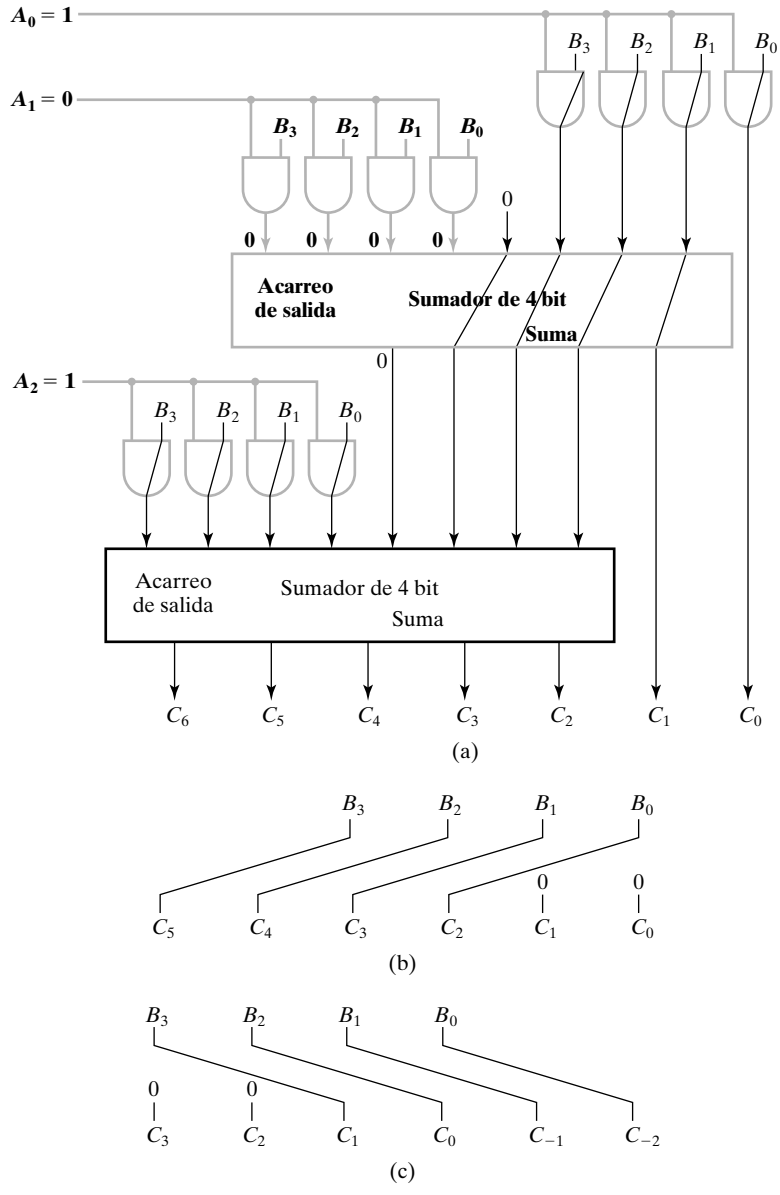
Multiplicación por constantes

Suponiendo que el circuito de la Figura 5-11 se emplee como base para la multiplicación, se consigue multiplicar por una constante simplemente aplicando la constante como el multiplicador *A*. Si el valor para una determinada posición de bits es 1, entonces el multiplicando se aplicará al sumador. Si el valor para una posición de bit es 0, entonces se aplicarán 0 al sumador y el sumador se eliminará por contracción. En ambos casos, las puertas AND se quitarán. El proceso se ilustra en la Figura 5-13(a). Para este caso, el multiplicador se ha puesto a 101. En la contracción, puesto que $0 + B = B$, el valor del acarreo de salida es siempre 0. El resultado final de la contracción es un circuito que lleva los dos bits de menor peso de B a las salidas C_1 y C_0. El circuito suma los dos bits más significativo de B a B desplazando dos posiciones a la izquierda y aplica el resultado del producto a las salidas desde C_6 hasta C_2.

Se produce un caso especialmente importante cuando la constante es igual 2^i (es decir, para la multiplicación $2^i \times B$). En este caso, sólo aparece un 1 en el multiplicador y toda la lógica se elimina, convirtiendo el circuito únicamente en cables. En este caso, para un 1 en la posición *i*, el resultado es *B* seguido por *i* 0. El bloque funcional que resulta es simplemente una combinación de desplazamientos y relleno a 0. La función de este bloque se denomina *desplazamiento a izquierda de i posiciones con relleno a ceros*. El *relleno a ceros* se refiere a añadir 0 a la derecha (o a la izquierda) de un operando como *B*. El desplazamiento es una operación muy importante aplicada tanto a datos numéricos como no numéricos. El resultado de contraer una multiplicación por 2^2 (es decir, un desplazamiento a izquierdas de 2 posiciones de bits) se muestra en Figura 5-13(b).

División por constantes

Puesto que no hemos tratado la operación de división, nuestra discusión sobre la división por constantes se limitará a la división por las potencias de 2 (es decir, por 2^i en binario). Dado que la multiplicación por 2^i resulta de añadir *i* 0 a la derecha del multiplicando, por analogía, la división por 2^i resulta de eliminar los *i* bits menos significativos del dividendo. Los bits restantes son el cociente y los bits desechados son el resto. La función de este bloque se denomina *desplazamiento a derechas de i posiciones*. Del mismo modo que el desplazamiento a izquierdas, el desplazamiento a derechas es igualmente una operación muy importante. El bloque fun-



□ **FIGURA 5-13**
Contracción de un multiplicador: (a) para 101 × *B*, (b) para 100 × *B* y (c) para *B* ÷ 100

cional para la división por 2^2 (es decir, desplazamiento a derechas de 2 posiciones) se muestra en Figura 5-13(c).

Relleno a ceros y extensión

El relleno a ceros, como se ha definido previamente para la multiplicación por una constante, también puede usarse para aumentar el número de bits de un operando. Por ejemplo, suponga que el byte 01101011 es la entrada de un circuito que necesita una entrada de 16 bits. Una ma-

nera posible de generar la entrada de 16 bits es rellenar a ceros por la izquierda con ocho 0 para producir 0000000001101011. Otra opción es rellenar a ceros por la derecha para producir 0110101100000000. El primer planteamiento sería apropiado para operaciones como la suma o la resta. La última aproximación podría usarse para efectuar una multiplicación de 16 bits con baja precisión, en la que el byte más significativo representa los 8 bits más significativos del resultado, mientras que el byte menos significativo es ignorado.

Frente al relleno a ceros, la *extensión del signo* se emplea para aumentar el número de bits en un operando usando la representación en complemento para los números con signo. Si el operando es positivo, entonces los bits pueden ser agregados a la izquierda extendiendo el signo del número (0 para positivo y 1 para negativo). El byte 01101011 que representa 107 en decimal, extendido a 16 bits se vuelve 0000000001101011. El byte 10010101, que en complemento a 2 representa -107, extendido a 16 bits se vuelve 1111111110010101. La razón para utilizar la extensión del signo es conservar la representación en complemento para los números con signo. Por ejemplo, si 10010101 estuviera extendido con 0, la magnitud representada sería muy grande y extensa, y el bit del extremo izquierdo que debería ser un 1 para un signo menos sería incorrecto en la representación en complemento a 2.

ARITMÉTICA DECIMAL Un suplemento que discute las operaciones aritméticas decimales y sus implemetaciones circuitales está disponible en la página web del libro.

5-7 REPRESENTACIONES HDL-VHDL

Hasta ahora, todas las descripciones usadas de VHDL contenían sólo una única entidad. Las descripciones que representan circuitos que emplean jerarquía tienen entidades múltiples, una para cada elemento distinto de la jerarquía, tal y cómo se muestra en el próximo ejemplo.

EJEMPLO 5-7 VHDL jerárquico para un sumador de 4 bits con acarreo serie

Los ejemplos de las Figuras 5-14 y 5-15 recurren a tres entidades para construir la descripción jerárquica de un sumador de 4 bits con acarreo serie. El estilo empleado para las arquitecturas será una mezcla de descripción estructural y de flujo de datos. Las tres entidades son un semi-sumador, un sumador completo basado en semi-sumadores, y el propio sumador de 4 bits. La arquitectura de `half_adder` consiste en dos asignaciones de flujo de datos, una para `s` y otra para `c`. La arquitectura del `full_adder` usa `half_adder` como componente. En la suma, se declaran tres señales internas, `hs`, `hc`, y `tc`. Estas señales se aplican a dos semi-sumadores y también se emplean en una asignación del flujo de datos para construir el sumador completo de la Figura 5-4. En la entidad `adder_4`, simplemente se conectan juntos los cuatro componentes del sumador completo usando las señales de la Figura 5-5.

Observe que mientras `C0` y `C4` son entrada y salida respectivamente, sin embargo `C(0)` y `C(4)` son señales internas (es decir, ni entradas ni salidas). `C(0)` se asigna a `C0` y `C4` se asigna a `C(4)`. El uso de `C(0)` y `C(4)` separadamente de `C0` y `C4` no es esencial aquí, pero es útil para ilustrar una restricción de VHDL. Suponga que queremos añadir la detección de desbordamiento al sumador tal y como aparece en la Figura 5-9. Si `C(4)` no se define separadamente, entonces podríamos intentar escribir

```
V <= C(3) xor C4
```

En VHDL, esto es incorrecto. Una salida no puede usarse como señal interna. Entonces, hay que definir una señal interna para utilizar en lugar de `C4` (por ejemplo, `C(4)`) dando

```
V <= C(3) xor C(4)
```

```
-- Sumador de 4 bits: descripción jerárquica de flujo de datos/estructural
-- (véase Figuras 5-4 y 5-5 para los diagramas lógicos)
library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
  port (x, y : in std_logic;
        s, c : out std_logic);
end half_adder;

architecture dataflow_3 of half_adder is
  begin
    s <= x xor y;
    c <= x and y;
end dataflow_3;

library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
  port (x, y, z : in std_logic;
        s, c : out std_logic);
end full_adder;

architecture struc_dataflow_3 of full_adder is
  component half_adder
    port(x, y : in std_logic;
         s, c : out std_logic);
  end component;
  signal hs, hc, tc: std_logic;
  begin
    HA1: half_adder
      port map (x, y, hs, hc);
    HA2: half_adder
      port map (hs, z, s, tc);
    c <= tc or hc;
end struc_dataflow_3;

library ieee;
use ieee.std_logic_1164.all;
entity adder_4 is
  port(B, A : in std_logic_vector(3 downto 0);
       C0 : in std_logic;
       S : out std_logic_vector(3 downto 0);
       C4: out std_logic);
end adder_4;
```

□ **FIGURA 5-14**
Descripción jerárquica de flujo de datos/estructural de un sumador de 4 bits

```
architecture structural_4 of adder_4 is
  component full_adder
    port(x, y, z : in std_logic;
         s, c : out std_logic);
  end component;
  signal C: std_logic_vector(4 downto 0);
  begin
    Bit0: full_adder
      port map (B(0), A(0), C(0), S(0), C(1));
    Bit1: full_adder
      port map (B(1), A(1), C(1), S(1), C(2));
    Bit2: full_adder
      port map (B(2), A(2), C(2), S(2), C(3));
    Bit3: full_adder
      port map (B(3), A(3), C(3), S(3), C(4));
    C(0) <= C0;
    C4 <= C(4);
end structural_4;
```

□ **FIGURA 5-15**
Descripción jerárquica de flujo de datos/estructural de un sumador de 4 bits (continuación) ■

Descripción de comportamiento

El sumador de 4 bits proporciona una oportunidad para ilustrar la descripción de circuitos con niveles superiores al nivel lógico. Estos niveles de descripción se refieren como nivel de comportamiento o nivel de transferencia entre registros. Estudiaremos específicamente el nivel de transferencia entre registros en el Capítulo 7. Aún sin estudiar el nivel de transferencia entre registros, sin embargo, podemos mostrar una descripción del nivel de comportamiento.

EJEMPLO 5-8 VHDL de comportamiento de un sumador de 4 bits con acarreo serie

La Figura 5-16 muestra la descripción de comportamiento para el sumador de 4 bits. En la arquitectura de la entidad `adder_4_b`, la suma lógica se describe por una única declaración que usa + y &. El + representa la suma y el & representa una operación denominada *concatenación*. El operador de concatenación combina dos señales en una sola señal que tiene un número de bits igual a la suma de los números de bits de cada una de las señales originales. En el ejemplo, `'0' & A` representa la señal

`'0' A(3) A(2) A(1) A(0)`

con 1 + 4 = 5 señales. Observe que `'0'`, que aparece a la izquierda en la expresión de la concatenación, aparece a la izquierda en el listado de la señal. Por consistencia, todas las entradas de la suma se convierten en cantidades de 5 bits por concatenación, ya que la salida, incluyendo `C4`, es de 5 bits. Esta conversión no es necesaria, pero es un planteamiento seguro.

Dado que + no puede realizarse en el tipo `std_logic`, necesitamos un paquete adicional para definir la suma para el tipo `std_logic`. En este caso estamos usando `std_logic_arith`, un paquete presente en la librería `ieee`. Además, deseamos definir la suma específicamente sin signo, por lo que emplearemos la extensión `unsigned`. También ocurre que la concatenación en VHDL no puede usarse en el lado izquierdo de una sentencia de asignación. Para obtener `C4` y `S` como resultado de la suma, se declara una señal `sum` de 5 bits. Se asigna el resultado de la

```
-- Sumador de 4 bits: descripción de comportamiento
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adder_4_b is
  port(B, A : in std_logic_vector(3 downto 0);
       C0 : in std_logic;
       S : out std_logic_vector(3 downto 0);
       C4: out std_logic);
end adder_4_b;

architecture behavioral of adder_4_b is
signal sum : std_logic_vector(4 downto 0);
begin
  sum <= ('0' & A) + ('0' & B) + ("0000" & C0);
  C4 <= sum(4);
  S <= sum(3 downto 0);
end behavioral;
```

□ **FIGURA 5-16**
Descripción de comportamiento de un sumador completo de 4-bits

suma, incluyendo el acarreo de salida, a la señal `sum`. Por último, las dos sentencias siguientes son asignaciones adicionales sobre `sum` para obtener las salidas `C4` y `S`. ■

Esto completa nuestra introducción a VHDL para los circuitos aritméticos. Continuaremos con más VHDL presentando maneras de describir los circuitos secuenciales en el Capítulo 6.

5-8 REPRESENTACIONES HDL-VERILOG

Hasta ahora, todas las descripciones empleadas han contenido únicamente un solo módulo. Las descripciones que representan circuitos que usan jerarquía tienen varios módulos, uno para cada elemento distinto de la jerarquía, tal y cómo se muestra en el siguiente ejemplo.

EJEMPLO 5-9 Verilog jerárquico para un sumador acarreo serie de 4 bits

La descripción de la Figura 5-17 recurre a tres módulos para representar el diseño jerárquico para un sumador de 4 bits con acarreo serie. El estilo usado para los módulos será una mezcla de descripción estructural y descripción de flujo de datos. Los tres módulos son un semi-sumador, un sumador completo construido a partir de semi-sumadores, y el propio sumador de 4 bits.

El módulo `half_adder` consiste en dos asignaciones de flujo de datos una para `s` y otra para `c`. El módulo `full_adder` usa el `half_adder` como componente, como en la Figura 5-4. En el módulo `full_adder` se declaran tres cables internos, `hs`, `hc`, y `tc`. Se aplican las entradas, salidas y estos cables a los dos semi-sumadores y mediante la OR de `tc` y `hc` se forma el acarreo `c`. Note que los mismos nombres pueden usarse en módulos diferentes (se usan por ejemplo, `x`, `y`, `s`, y `c` en `half_adder` y `full_adder`).

En el módulo `adder_4` simplemente se conectan juntos los cuatro sumadores completos mediante las señales de la Figura 5-5. Observe que `C0` y `C4` son entrada y salida respectivamente, mientras que `C(3)` a `C(1)` son señales internas (es decir, ni entradas ni salidas).

```
// Sumador de 4 bits: descripción jerárquica de flujo de datos/estructural
// (véase Figuras 5-4 y 5-5 para los diagramas lógicos)

module half_adder_v(x, y, s, c);
   input x, y;
   output s, c;

   assign s = x ^ y;
   assign c = x & y;

endmodule

 odule full_adder_v(x, y, z, s, c);
   input x, y, z;
   output s, c;

   wire hs, hc, tc;

   half_adder_v    HA1(x, y, hs, hc),
                   HA2(hs, z, s, tc);
   assign c = tc | hc;

endmodule

 odule adder_4_v(B, A, C0, S, C4);
   input[3:0] B, A;
   input C0;
   output[3:0] S;
   output C4;

   wire[3:1] C;

   full_adder_v    Bit0(B[0], A[0], C0, S[0], C[1]),
                   Bit1(B[1], A[1], C[1], S[1], C[2]),
                   Bit2(B[2], A[2], C[2], S[2], C[3]),
                   Bit3(B[3], A[3], C[3], S[3], C4);
endmodule
```

□ **FIGURA 5-17**
Descripción jerárquica de flujo de datos/estructural Verilog para un sumador de 4-bits ■

Descripción de comportamiento

El sumador de 4 bits proporciona una oportunidad para ilustrar la descripción de circuitos con niveles superiores a un nivel lógico. Tales niveles de descripción se denominan nivel de comportamiento o nivel de transferencia entre registros. Estudiaremos específicamente el nivel de transferencia entre registros en el Capítulo 7. Aún sin estudiar el nivel de transferencia entre registros, podemos mostrar sin embargo la descripción a nivel de comportamiento para el sumador de 4 bits.

EJEMPLO 5-10 Verilog de comportamiento para un sumador de 4 bits con acarreo serie

La Figura 5-18 muestra la descripción Verilog para el sumador. En el módulo `adder_4_b_v`, la suma lógica se describe mediante una única sentencia que usa + y `{}`. El + representa la suma y el `{}` representa una operación denominada *concatenación*. La operación + realizada sobre los tipos de datos del cable es sin signo. La concatenación combina dos señales en una sola

señal que tiene un número de bits igual a la suma del número de bits en las señales originales. En el ejemplo, `{C4, S}` representa el vector

`C4 S[3] S[2] S[1] S[0]`

con 1 + 4 = 5 señales. Observe que `C4` aparece a la izquierda en la expresión de la concatenación y aparece a la izquierda en el listado de la señal.

```
// Sumador de 4 bits: descripción Verilog de comportamiento

module adder_4_b_v(A, B, C0, S, C4);
   input[3:0] A, B;
   input C0;
   output[3:0] S;
   output C4;

   assign {C4, S} = A + B + C0;
endmodule
```

□ **FIGURA 5-18**
Descripción de comportamiento de un sumador completo de 4-bits usando Verilog ■

Esto completa nuestra introducción a Verilog para los circuitos aritméticos. En el Capítulo 6 se presentarán los mecanismos ofrecidos por Verilog para la descripción de circuitos secuenciales.

5-9 RESUMEN DEL CAPÍTULO

En este capítulo se han presentado circuitos que efectúan operaciones aritméticas. Se ha tratado en detalle la implementación de sumadores binarios, incluso del sumador con acarreo paralelo para mejorar el rendimiento. Se ha mostrado la resta de números binarios sin signo empleando el complemento a 2 y a 1, así como la representación de números binarios con signo, y su suma y su resta. El sumador-restador, desarrollado para los binarios sin signo también se ha utilizado para sumar y restar directamente números con signo en complemento a 2. Se ha realizado una breve introducción a la multiplicación binaria usando circuitos combinacionales compuestos de puertas AND y sumadores binarios.

Igualmente se han presentado operaciones aritméticas adicionales como el incremento, el decremento, la multiplicación y división por una constante, y los desplazamientos. Se obtuvieron aplicaciones para estas operaciones por medio de una técnica de diseño denominada contracción. También se han presentado el relleno con ceros y la extensión del signo del operando.

Las últimas dos secciones del capítulo han proporcionado una introducción a las descripciones de circuitos aritméticos en VHDL y Verilog. Ambos HDLs se han ilustrado estudiando las descripciones a nivel funcional y de comportamiento para varios bloques funcionales del capítulo.

REFERENCIAS

1. MANO, M. M.: *Digital Design*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2002.
2. WAKERLY, J. F.: *Digital Design*: *Principles and Practices*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2000.
3. *High-Speed CMOS Logic Data Book*. Dallas: Texas Instruments, 1989.

4. *IEEE Standard VHDL Language Reference Manual*. (ANSI/IEEE Std 1076- 1993; revision of IEEE Std 1076-1987). New York: The Institute of Electrical and Electronics Engineers, 1994.
5. SMITH, D. J.: *HDL Chip Design*. Madison, AL: Doone Publications, 1996.
6. PELLERIN, D. and D. TAYLOR: *VHDL Made Easy!* Upper Saddle River, NJ: Prentice Hall PTR, 1997.
7. STEFAN, S. and L. LINDH: *VHDL for Designers*. London: Prentice Hall Europe, 1997.
8. *IEEE Standard Description Language Based on the Verilog® Hardware Description Language* (IEEE Std 1364-1995). New York: The Institute of Electrical and Electronics Engineers, 1995.
9. PALNITKAR, S.: *Verilog HDL: A Guide to Digital Design and Synthesis*. Upper Saddle River, NJ: SunSoft Press (A Prentice Hall Title), 1996.
10. BHASKER, J.: *A Verilog HDL Primer*. Allentown, PA: Star Galaxy Press, 1997.
11. THOMAS, D., and P. MOORBY: *The Verilog Hardware Description Language* 4th ed. Boston: Kluwer Academic Publishers, 1998.
12. CILETTI, M.: *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL*, Upper Saddle River, NJ: Prentice Hall, 1999.

PROBLEMAS

El símbolo (+) indica problemas más avanzados y el asterisco (*) indica que la solución se puede encontrar en el sitio web del libro: http://www.librosite.net/Mano.

5-1. Diseñe un circuito combinacional que obtenga la suma binaria en 2 bits S_1S_0 de dos números de 2 bits A_1A_0 y B_1B_0 y que tiene una entrada de acarreo C_0 y una salida de acarreo C_2. Diseñe el circuito entero que implemente cada una de las tres salidas como un circuito de dos niveles más inversores para las variables de entrada. Empiece el diseño con las siguientes ecuaciones para cada uno de los dos bits del sumador:

$$S_i = \overline{A}_i\overline{B}_iC_i + \overline{A}_iB_i\overline{C}_i + A_i\overline{B}_i\overline{C}_i + A_iB_iC_i$$
$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$$

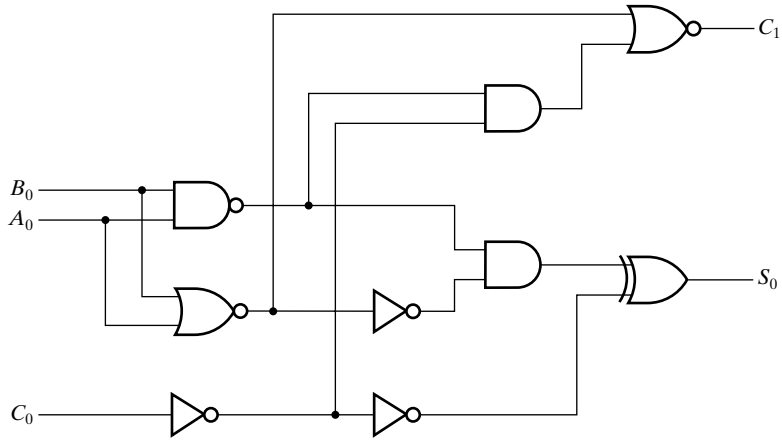
5-2. *El diagrama lógico de la primera etapa de un sumador de 4 bits, como el implementado en el circuito integrado 74283, se muestra en la Figura 5-19. Verifique que el circuito implementa un sumador completo.

5-3. *Obtenga los complementos a 1 y a 2 de los siguientes números binarios sin signo: 10011100, 10011101, 10101000, 00000000, y 10000000.

5-4. Realice la resta indicada con los siguientes números binarios sin signo tomando el complemento a 2 del substraendo:

(a) 11111 − 10000 **(c)** 1011110 − 1011110
(b) 10110 − 1111 **(d)** 101 − 101000

5-5. Repita el Problema 5-4 suponiendo que los números están en complemento a 2 con signo. Use la extensión para igualar la longitud de los operandos. ¿Se produce desbordamiento durante las operaciones del complemento para cualquiera de los números dados? ¿Ocurre desbordamiento en la resta global para cualquiera de los números dados?



□ **FIGURA 5-19**
Circuito del Problema 5-2

5-6. *Calcule las operaciones aritméticas $(+36) + (-24)$ y $(-35) - (-24)$ en binario empleando la representación en signo y complemento a 2 para los números negativos.

5-7. Los siguientes números binarios tienen un signo en la posición del extremo izquierdo y, si son negativos, están en complemento a 2. Realice las operaciones aritméticas indicadas y verifique las respuestas.

(a) 100111 + 111001 **(c)** 110001 − 010010
(b) 001011 + 100110 **(d)** 101110 − 110111

Indique si hay desbordamiento para cada cálculo.

5-8. +Diseñe tres versiones de un circuito combinacional cuya entrada es un número de 4 bits y cuya salida es el complemento a 2 del número de la entrada, para cada uno de los siguientes casos:

(a) El circuito es un circuito simplificado a dos niveles, más los inversores necesarios para las variables de entrada.
(b) El circuito está formado por cuatro células idénticas de dos entradas y dos salidas, una para cada bit. Las células se conectan en cascada, con líneas entre ellas similares a un acarreo. El valor aplicado al bit de acarreo final derecho es 0.
(c) El circuito se rediseña con lógica de acarreo anticipado para aumentar la velocidad del circuito del apartado b, de modo que pueda emplearse en circuitos más grandes con $4n$ bits de entrada.

5-9. Emplee contracción a partir de un sumador de 4 bits con acarreo para diseñar el circuito de un incrementador por 2 de 4 bits con acarreo que añade el valor binario 0010 a su entrada de 4 bits. La función a implementar es $S = A + 0010$.

5-10. Use contracción partiendo de un sumador-restador de 8 bits sin acarreo para diseñar un circuito de 8 bits sin acarreo que incrementa su entrada en 00000010 cuando la entrada $S = 0$ y disminuye su entrada en 00000010 cuando la entrada $S = 1$. Realice el diseño diseñando las distintas células de 1 bit necesarias e indicando qué tipo de célula emplear en cada una de las ocho posiciones de bits.

5-11. + **(a)** Usando contracción a partir de un sumador de 4 bits con acarreo anticipado, con entrada de acarreo y funciones de propagación de grupo y acarreo de grupo diseñe un circuito de 4 bits con acarreo paralelo que incrementa sus 4 bits de entrada en el equivalente binario de 2.

(b) Repita el apartado **(a)**, diseñando un circuito con acarreo anticipado que sume 0000 a su entrada de 4 bits.

(c) Construya un circuito de 16 bits con acarreo anticipado que incremente su entrada de 16 bits en 2, dando la salida de 16 bits más un acarreo de salida. Emplee una contracción adicional sobre los circuitos diseñados en los apartados **(a)** y **(b)** para producir C_4, C_8, y C_{12} y G_{0-15} y P_{0-15} con las entradas P_{0-3}, G_{0-3}, P_{4-7}, P_{8-11}, y P_{12-15}, y la lógica adicional necesaria para producir C_{16}. (Observe que, debido al apartado **(b)**, G_{4-7}, G_{8-11}, y G_{12-15} son iguales a 0.)

5-12. Diseñe un circuito combinacional que compare dos números sin signo de 4 bits A y B para ver si B es mayor que A. El circuito tiene una salida X, de modo que $X = 1$ si $A < B$ y $X = 0$ si $A = B$.

5-13. + Repita el Problema 5-12 usando circuitos de tres entradas y una salida, uno para cada uno de los cuatro bits. Los cuatro circuitos se conectan juntos en cascada a través de las señales de acarreo. Una de las entradas a cada célula es una entrada de acarreo, y la única salida es la del acarreo.

5-14. Repita el Problema 5-12 aplicando contracción a un restador de 4 bits y usando el acarreo externo como X.

5-15. Diseñe un circuito combinacional que compare los números sin signo de 4 bits A y B para ver si $A = B$ o $A < B$. Use un circuito iterativo como en el Problema 5-14.

5-16. + Diseñe un sumador-restador en signo y magnitud de 5 bits. Divida el circuito para el diseño en (1) la generación del signo y la lógica de control del sumador-restador, (2) un sumador-restador de números sin signo que emplea el complemento a 2 del minuendo para la resta, y (3) la corrección lógica selectiva en complemento a 2 del resultado.

5-17. *El circuito sumador-restador de la Figura 5-8 tiene los siguientes valores para la entrada de selección S y para las entradas de datos A y B:

	S	***A***	***B***
(a)	0	0111	0111
(b)	1	0100	0111
(c)	1	1101	1010
(d)	0	0111	1010
(e)	1	0001	1000

Determine, en cada caso, los valores de las salidas S_3, S_2, S_1, S_0 y C_4.

5-18. *Diseñe un multiplicador binario que multiplique dos números sin signo de 4 bits. Emplee puertas AND y sumadores binarios.

5-19. Diseñe un circuito que multiplique un número de 4 bits por la constante 1010 aplicando contracción a la solución del Problema 5-18.

5-20. **(a)** Diseñe un circuito que multiplique un dato de 4 bits por la constante 1000.
(b) Diseñe un circuito que divida un dato de 8 bits entre la constante 1000 dando un cociente de 8 bits y un resto de 8 bits.

Todos los archivos HDL para circuitos referidos en los restantes problemas están disponibles en ASCII para su simulación y edición en el sitio web del libro. Para los problemas que piden simulación se necesita un compilador/simulador de VHDL o Verilog. En cualquier caso, siempre se pueden escribir las descripciones HDL de muchos problemas sin necesidad de compilar o simular.

5-21. Compile y simule el sumador de 4 bits de las Figuras 5-14 y 5-15. Aplique estímulos que verifiquen el sumador completo más a la derecha para las ocho posibles combinaciones de entrada; esto también sirve como chequeo para los otros sumadores completos. Aplique también combinaciones que verifiquen las conexiones de la cadena de acarreos entre todos los sumadores completos demostrando que puede propagarse un 0 y un 1 desde `C0` hasta `C4`.

5-22. *Compile y simule la descripción de comportamiento del sumador de 4 bits de la Figura 5-16. Suponiendo una implementación con acarreo serie, aplique combinaciones que comprueben el sumador completo de más a la derecha para las ocho combinaciones de entrada. Aplique también combinaciones que verifiquen las conexiones de la cadena de acarreos entre todos los sumadores demostrando que puede propagarse un 0 y un 1 de `C0` a `C4`.

5-23. +Usando la Figura 5-16 como guía y un *when else* para `S`, escriba la descripción de comportamiento VHDL en alto nivel para el sumador-restador de la Figura 5-8. Compile y simule su descripción. Suponiendo una implementación con acarreo serie, aplique combinaciones que comprueben una de las etapas del sumador-restador para las 16 posibles combinaciones de entrada. También, aplique las combinaciones para verificar las conexiones de la cadena de acarreos entre los sumadores completos demostrando que pueden propagarse un 0 y un 1 de `C0` a `C4`.

5-24. +Escriba la descripción jerárquica de flujo de datos en VHDL similar a la de las Figuras 5-14 y 5-15 para la lógica del sumador de 4-bits con acarreo anticipado de la Figura 5-6. Compile y simule su descripción. Obtenga un conjunto de vectores de test que permitan verificar de forma convincente la lógica del sumador.

5-25. Compile y simule el sumador de 4 bits de la Figura 5-17. Aplique combinaciones que comprueben el sumador completo más a la derecha para las ocho posibles combinaciones de entrada; esto también servirá como test para los otros sumadores completos. Aplique también combinaciones que verifiquen las conexiones de la cadena de acarreos entre todos los sumadores demostrando que puede propagarse un 0 y un 1 de `C0` a `C4`.

5-26. *Compile y simule la descripción de comportamiento del sumador de 4 bits de la Figura 5-18. Asumiendo una implementación de acarreo serie, aplique las ocho combinaciones de entrada para comprobar el sumador completo más a la derecha. Aplique también combinaciones que verifiquen las conexiones de la cadena de acarreos entre todos los sumadores completos, demostrando que puede propagarse un 0 y un 1 de `C0` a `C4`.

5-27. Usando la Figura 5-18 como guía y con una «decisión binaria» en `S` de la Figura 4-37, escriba la descripción de comportamiento de alto nivel en Verilog para el sumador-restador de la Figura 5-8. Compile y simule su descripción. Suponiendo una implementa-

ción de acarreo serie, proponga un conjunto de vectores de test que (1) apliquen las 16 posibles combinaciones de entrada a la etapa sumadora-restadora del bit 2 y (2) obtengan el acarreo de salida de esta etapa a través de alguna salida del circuito. Aplique también combinaciones que permitan verificar las conexiones de la cadena de acarreo entre todos los sumadores completos, demostrando que puede propagarse un 0 y un 1 de `C0` a `C4`.

5-28. + Escriba una descripción de flujo de datos jerárquica en Verilog similar a la de la Figura 5-17 para el sumador de 4 bits con acarreo anticipado de la Figura 5-6. Compile y simule su descripción. Localice un conjunto de vectores de test que permitan verificar de forma convincente el funcionamiento del circuito.

CAPÍTULO

6

CIRCUITOS SECUENCIALES

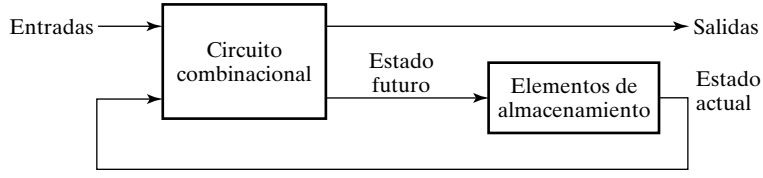
Hasta este momento hemos estudiado únicamente lógica combinacional. Aunque esta lógica es capaz de realizar interesantes operaciones tales como la suma y la resta, la realización de secuencias de operaciones empleando lógica combinacional requiere una cascada de muchas estructuras juntas. Sin embargo, el hardware para lograrlo es muy costoso y poco flexible. Para conseguir secuencias de operaciones útiles y flexibles, necesitamos poder construir circuitos que permitan guardar la información entre las operaciones. Tales circuitos son los denominados circuitos secuenciales. Este capítulo empieza con una introducción a los circuitos secuenciales seguida por un estudio de los elementos básicos para almacenar información binaria, denominados latches y flip-flops. Distinguiremos los latches de los flip-flops y estudiaremos varios tipos de cada uno. Más tarde analizaremos los circuitos secuenciales consistentes en flip-flops y lógica combinacional. Las tablas y los diagramas de estado proporcionan una forma de describir el comportamiento de los circuitos secuenciales. Las siguientes secciones del capítulo desarrollan técnicas para diseñar circuitos secuenciales y verificar su correcto funcionamiento. En las dos últimas secciones, con los lenguajes de descripción hardware VHDL y Verilog proporcionamos representaciones de los elementos de almacenamiento para los tipos de circuitos secuenciales de este capítulo.

Los latches, los flip-flops y los circuitos secuenciales son componentes fundamentales en el diseño de la mayoría de la lógica digital. En la computadora genérica mostrada al principio del Capítulo 1, los latches y los flip-flops están muy extendidos en todo el diseño. La excepción son los circuitos de memoria, puesto que se diseñan grandes cantidades de memoria como circuitos electrónicos en lugar de como circuitos lógicos. No obstante, debido al empleo masivo del almacenamiento basado en lógica, este capítulo contiene material fundamental para que cualquiera entienda en profundidad las computadoras y los sistemas digitales y cómo se diseñan.

6–1 DEFINICIÓN DE CIRCUITO SECUENCIAL

Los circuitos digitales que se han considerado hasta ahora han sido los combinacionales. Aunque la mayoría de los sistemas digitales incluyen circuitos combinacionales, la mayoría de los sistemas encontrados en la práctica incluyen también elementos de almacenamiento por lo que el sistema se describe como un sistema secuencial.

En la Figura 6-1 se muestra el diagrama de bloques de un circuito secuencial. Para formar un circuito secuencial se interconecta un circuito combinacional con elementos de almacenamiento. Los elementos de almacenamiento son circuitos capaces de almacenar información binaria. La información binaria guardada en estos elementos en un momento dado define el estado del circuito secuencial en ese momento. El circuito secuencial recibe la información binaria de su entorno a través de las entradas. Estas entradas, junto con el estado actual de los elementos de almacenamiento, determinan el valor binario de las salidas. También determinan los valores empleados para determinar el próximo estado de los elementos de almacenamiento. El diagrama de bloques demuestra que las salidas de un circuito secuencial no sólo son función de las entradas, sino también del estado actual de los elementos de almacenamiento. El próximo estado de los elementos de almacenamiento también es una función de las entradas así como del estado presente. De este modo, un circuito secuencial se especifica por una sucesión en el tiempo de entradas, estados interiores, y salidas.

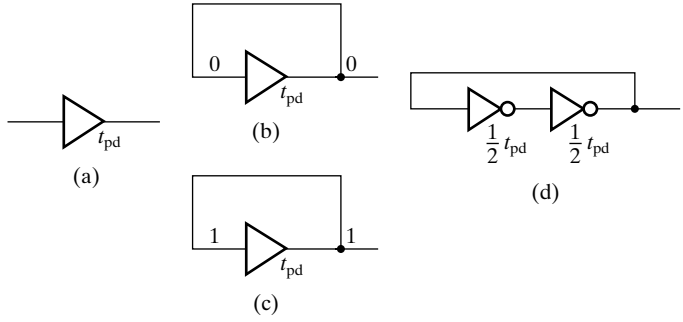


□ **FIGURA 6-1**
Diagrama de bloques de un circuito secuencial

Hay dos tipos principales de circuitos secuenciales, y su clasificación depende del momento en que se observan sus entradas y de los cambios de su estado interior. El comportamiento de un circuito secuencial síncrono se define a partir del conocimiento de sus señales en instantes discretos de tiempo. El comportamiento de un circuito secuencial asíncrono depende tanto de las entradas en cualquier instante de tiempo como del orden en que cambian las entradas a lo largo del tiempo.

La información se almacena en los sistemas digitales de muchas maneras, incluso mediante el empleo de circuitos lógicos. La Figura 6-2(a) muestra un buffer. Este buffer tiene un retardo de propagación t_{pd}. Puesto que la información presente en el buffer de entrada en un instante de tiempo t aparece en el buffer de salida en el instante $t + t_{pd}$, la información se ha almacenado eficazmente durante un tiempo t_{pd}. Pero, en general, deseamos almacenar la información durante un tiempo indefinido típicamente mayor que el tiempo de retardo de una o incluso muchas puertas. Este valor almacenado cambiará en determinados momentos en base a las entradas que se apliquen al circuito y no dependerá del tiempo de retardo específico de una puerta.

Suponga que la salida del buffer de la Figura 6-2(a) se conecta a su entrada tal y como muestran las Figuras 6-2(b) y (c). Además, suponga que el valor en la entrada del buffer en el caso (b) ha sido 0 durante por lo menos un tiempo t_{pd}. Entonces la salida producida por el buffer será 0 tras un tiempo $t + t_{pd}$. Esta salida se aplica a la entrada de modo que la salida volverá a ser 0 en un tiempo $t + 2t_{pd}$. Esta relación entre la entrada y la salida se mantiene para todo t, de



□ **FIGURA 6-2**
Estructuras lógicas para almacenamiento de información

modo que los 0 se almacenarán indefinidamente. Puede emplearse el mismo argumento para almacenar un 1 en el circuito de la Figura 6-2(c).

El ejemplo del buffer muestra cómo construir un almacenamiento a partir de lógica con retardo conectada en lazo cerrado. Cualquier lazo que produzca dicho almacenamiento deberá compartir con el buffer una propiedad, a saber, que no debe haber ninguna inversión alrededor del lazo. Normalmente, un buffer se implementa con dos inversores, como se muestra en la Figura 6-2(d). La señal se invierte dos veces, es decir,

$$\overline{\overline{X}} = X$$

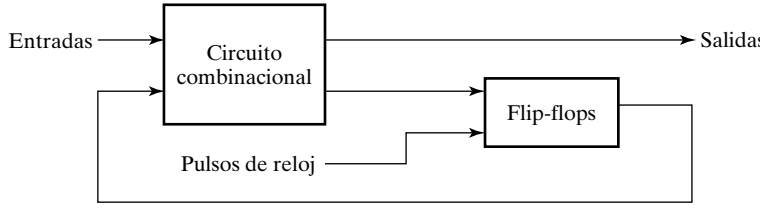
no originando ninguna inversión alrededor de todo el lazo. En efecto, este ejemplo ilustra uno de los métodos más populares de implementar el almacenamiento en las memorias del ordenador. (*Véase* Capítulo 9.) Sin embargo, aunque los circuitos de las Figuras 6-2(b) hasta (d) pueden almacenar información, no hay ningún modo para poder variarla. Reemplazando los inversores por puertas NOR o NAND, la información puede cambiarse. Los circuitos de almacenamiento asíncronos denominados latches están realizados de este modo y se discuten en la próxima sección.

En general, los circuitos asíncronos más complejos son difíciles de diseñar, puesto que su comportamiento depende mucho del retardo de propagación de las puertas y del instante en que cambian las entradas. Así, la mayoría de los diseñadores optan por circuitos que se ajustan al modelo síncrono. No obstante, algunas veces se necesitan diseños asíncronos. Un caso muy importante es el empleo de latches asíncronos como bloques para construir elementos de almacenamiento, denominados flip-flops, que guardan la información en circuitos síncronos.

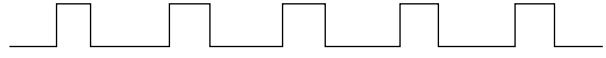
Un circuito secuencial síncrono emplea señales que sólo afectan a los elementos de almacenamiento en momentos determinados de tiempo. La sincronización se logra mediante un dispositivo de temporización denominado *generador de reloj* que produce un tren periódico de pulsos de reloj. Los pulsos se distribuyen a lo largo del sistema de tal manera que los elementos de almacenamiento síncronos sólo están afectados por determinados pulsos. En la práctica, los pulsos del reloj se aplican junto con otras señales que indican cuándo deben cambiar los elementos de almacenamiento. El valor de las salidas de los elementos de almacenamiento sólo puede cambiar en presencia de pulsos de reloj. Los circuitos secuenciales síncronos que emplean pulsos de reloj como entrada a los elementos de almacenamiento se denominan *circuitos secuenciales síncronos*. Este tipo de circuitos es el más frecuentemente empleado, puesto que funcionan correctamente a pesar de las grandes diferencias en los retardos del circuito y porque son relativamente fáciles de diseñar.

Los elementos de almacenamiento empleados en los circuitos secuenciales con entrada de reloj se denominan flip-flops. Por simplicidad, suponga una única entrada de señal de reloj por circuito. Un flip-flop es un dispositivo de almacenamiento binario capaz de guardar un bit de información y que presenta los parámetros temporales que se definirán en la Sección 6-3. El diagrama de bloques de un circuito secuencial síncrono se muestra en la Figura 6-3. Los flip-flops reciben sus entradas del circuito combinacional y también de una señal de reloj con pulsos de frecuencia fija —que ocurren a intervalos fijos de tiempo—, tal y como se muestra en el diagrama de tiempos. Los flip-flops sólo pueden cambiar en respuesta a un pulso de reloj. Para el funcionamiento síncrono, en ausencia de señal de reloj, las salidas del biestable no cambian aun cuando las salidas del circuito combinacional que controla sus entradas hayan cambiado de valor. Así, el lazo de realimentación que se aprecia en la figura entre el circuito combinacional y el flip-flop está roto. Como resultado, la transición de un estado a otro sólo ocurre a intervalos de tiempo fijos dictados por los pulsos de reloj, dando el funcionamiento síncrono. Las salidas del circuito secuencial se muestran como las salidas del circuito combinacional. Esto es válido incluso cuando algunas salidas del circuito secuencial son realmente las salidas de los biestables. En este caso, la parte del circuito combinacional entre las salidas de los flip-flops y las salidas del circuito secuencial consiste sólo en conexiones.

Un flip-flop tiene una o dos salidas, una para el valor normal del bit almacenado y otra optativa para el valor complementado del bit almacenado. La información binaria puede entrar en un flip-flop de muy variadas maneras, hecho que da lugar a los diferentes tipos de flip-flops. Nuestro enfoque esta centrado en el flip-flop más empleado en nuestros días, el flip-flop D. En la Sección 6-6, se considerarán otros tipos de flip-flops. Como adelanto al estudio de los flip-flops y de su funcionamiento, en la próxima sección se presentan conocimientos básicos sobre los latches empleados para construir los flip-flops.



(a) Diagrama de bloque



(b) Cronograma de los pulsos de reloj

□ **FIGURA 6-3**
Circuito secuencial síncrono con reloj

6-2 LATCHES

Un elemento de almacenamiento puede mantener un estado binario indefinidamente (mientras no se retire la alimentación del circuito), hasta que una señal de entrada decida cambiar su estado. La mayor diferencia entre los distintos tipos de latches y flip-flops está en el número de entradas que poseen y en la manera en que las entradas afectan al estado binario. Los elementos de almacenamiento más básicos son los latches con los que normalmente se suelen construir los flip-flops. Aunque a menudo se emplean latches dentro de los flip-flops, también se pueden usar

métodos de sincronización más complejos para implementar directamente circuitos secuenciales. Sin embargo, el diseño de dichos circuitos va más allá del alcance del tratamiento básico que damos aquí. En esta sección, el enfoque está en los latches como primitivas básicas para construir elementos de almacenamiento.

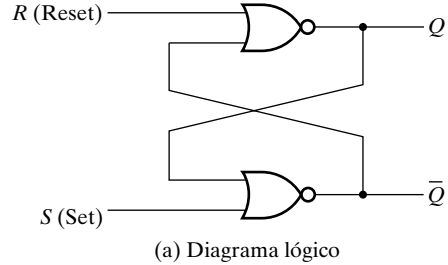
Latches *RS* y $\bar{R}\bar{S}$

El biestable o latch *RS* es un circuito construido de dos puertas NOR contrapuestas. Se obtiene a partir del elemento de almacenamiento de lazo sencillo de la Figura 6-2(d) sin más que reemplazar los inversores por puertas NOR, tal y como se muestra en la Figura 6-4(a). Esta modificación permite que el valor almacenado en el latch pueda cambiar. El latch tiene dos entradas etiquetadas como *S* (por *set*) y *R* (por *reset*), y dos estados. Cuando la salida $Q = 1$ y $\bar{Q} = 0$, se dice que el biestable está en el *estado SET*. Cuando $Q = 0$ y $\bar{Q} = 1$, está en el *estado RESET*. Las salidas Q y $\bar{Q}$ son normalmente la una el complemento de la otra. Cuando ambas entradas son iguales a 1 al mismo tiempo, se produce un estado indefinido con ambas salidas igual a 0.

En condiciones normales, las dos entradas del latch permanecen a 0 a menos que deseemos que el estado cambie. La aplicación de un 1 en la entrada *S* lleva al latch a colocarse en el estado set (1). La entrada *S* debe volver a 0 antes de que *R* cambie a 1 para evitar que se llegue al estado indefinido. Tal y como se aprecia en la tabla de funcionamiento de la Figura 6-4(b), hay dos condiciones de entrada al circuito que causan el estado set. La condición inicial es $S = 1$, $R = 0$, que lleva al circuito al estado set. Aplicar un 0 en *S* con $R = 0$ coloca al circuito en el mismo estado set. Después de que ambas entradas vuelvan a 0, es posible colocar el estado reset aplicando un 1 a la entrada *R*. Entonces, podemos retirar el 1 de *R*, y el circuito permanecerá en el estado reset. De este modo, cuando ambas entradas son iguales a 0, el latch puede estar en set o en reset, dependiendo en qué entrada se colocó más recientemente un 1.

Si se aplica un 1 en ambas entradas del latch, las dos salidas Q y $\bar{Q}$ son 0. Esto produce un estado indefinido porque viola el requisito de que las salidas son el complemento la una de la otra. Cuando ambas entradas devuelven un 0 simultáneamente también se produce un próximo estado indeterminado o imprevisible. En el funcionamiento normal, estos problemas se evitan asegurándonos que no se aplican 1 simultáneamente en ambas entradas.

El comportamiento del latch *RS* descrito en el párrafo anterior se ilustra en la Figura 6-5 mediante el simulador lógico de formas de onda ModelSim. Inicialmente, se desconocen las entradas así como el estado del latch, indicado con un nivel lógico a medio camino entre un nivel lógico 0 y 1. Cuando *R* pasa a ser 1 con *S* a 0, el latch se pone a 0, primero *Q* se hace 0 y, en respuesta, *Q_b* (que representa $\bar{Q}$) se hace 1. Después, cuando *R* cambia a 0, el latch permanece

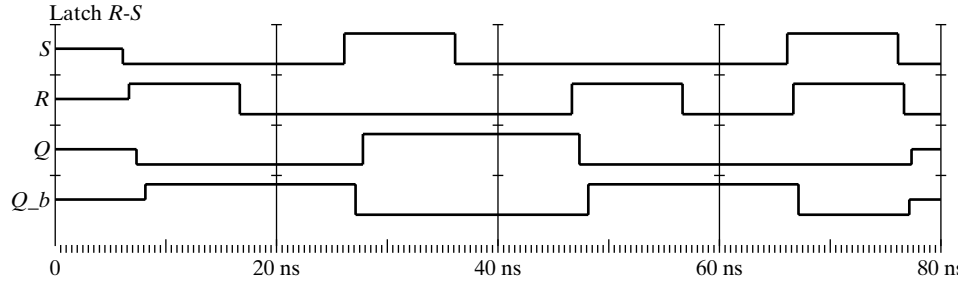


(a) Diagrama lógico

S	R	Q	$\bar{Q}$	
1	0	1	0	Estado «Set»
0	0	1	0	
0	1	0	1	Estado «Reset»
0	0	0	1	
1	1	0	0	Indefinido

(b) Tabla de funcionamiento

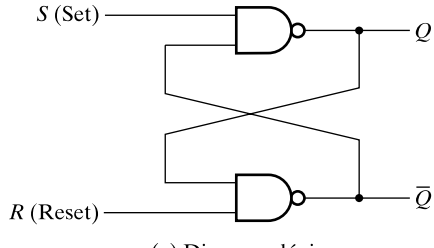
□ **FIGURA 6-4**
Latch *SR* con puertas NOR



□ **FIGURA 6-5**
Simulación lógica del comportamiento del latch *SR*

a reset almacenando el valor 0 presente en *Q*. Cuando *S* pasa a 1 con *R* a 0, el latch se coloca en set, primero *Q_b* se pone a 0 y, en respuesta, después *Q* se pone a 1. Los retardos en los cambios de *Q* y *Q_b* tras un cambio en la entrada están directamente relacionados con los retardos de las dos puertas NOR empleadas en la implementación del latch. Cuando *S* vuelve a ser 0, el latch se mantiene en set, almacenando el valor 1 presente en *Q*. Cuando *R* pasa a ser 1 con *S* igual a 0, el latch se resetea de modo que *Q* cambia a 0 y *Q_b* responde cambiando a 1. El latch permanece en reset cuando *R* vuelve a 0. Cuando *S* y *R* pasan a ser ambos 1, *Q* y *Q_b* toman el valor 0. Cuando *S* y *R* simultáneamente retornan 0, *Q* y *Q_b* toman valores desconocidos. Esta forma de comportamiento con estados indeterminados para la secuencia de entradas ($\overline{S}$, $\overline{R}$): (1, 1), (0, 0) resulta de suponer cambios simultáneos en las entradas y retardos de puerta iguales. El verdadero comportamiento indeterminado que se produce es debido a los retardos del circuito y de ligeras diferencias en los tiempos en que cambian *S* y *R* en el circuito real. Sin tener en cuenta los resultados de la simulación, estos comportamientos indeterminados se ven como indeseables, y se trata de evitar la combinación de entrada (1, 1). En el general, los cambios de estado del latch sólo responden a cambios en las entradas, permaneciendo inalterado el resto del tiempo.

La Figura 6-6 muestra el latch $\overline{S}\overline{R}$ con dos puertas NAND contrapuestas. Normalmente funciona con ambas entradas a 1, a menos que el estado del latch tenga que cambiar. La aplicación de un 0 en *S* provoca que la salida *Q* se coloque a 1, poniendo el latch en el estado set. Cuando la entrada *S* se vuelve 1, el circuito permanece en el estado set. Con ambas entradas a 1, el estado del latch se puede cambiar poniendo un 0 en la entrada *R*. Esto lleva al circuito al estado reset permaneciendo así aun después de que ambas entradas vuelvan a retomar el valor 1. Para este latch NAND, la condición indefinida se produce cuando ambas entradas son iguales a 0 al mismo tiempo, una combinación de entradas que debe evitarse.



(a) Diagrama lógico

S	R	Q	$\overline{Q}$	
0	1	1	0	Estado «Set»
1	1	1	0	
1	0	0	1	Estado «Reset»
1	1	0	1	
0	0	1	1	Indefinido

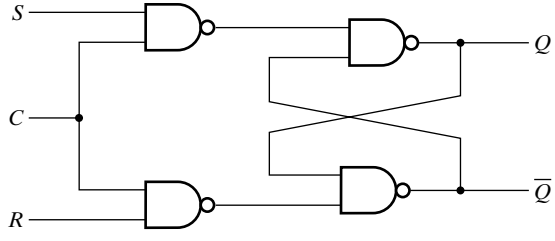
(b) Tabla de funcionamiento

□ **FIGURA 6-6**
Latch $\overline{S}\overline{R}$ con puertas NAND

Comparando el latch NAND con el NOR, observe que las señales de entrada para el NAND son las complementarias de las empleadas para el NOR. El latch NAND se denomina $\overline{S}\overline{R}$ porque exige una señal 0 para cambiar su estado. La barra sobre las letras indica el hecho de que las entradas deben estar complementadas para actuar sobre el estado del circuito.

El funcionamiento de los latches básicos NOR y NAND puede modificarse añadiendo una entrada de control adicional que determina cuándo puede cambiar el estado del latch. En la Figura 6-7 se muestra un latch *SR* con una entrada de control. Consiste en un latch NAND básico y dos puertas NAND adicionales. La entrada de control *C* actúa como una señal habilitadora para las otras dos entradas. Las salidas de las puertas NAND permanecen en el nivel lógico 1 mientras que la entrada de control esté a 0. Ésta es la condición de mantenimiento del dato para el latch compuesto por dos puertas NAND. Cuando la entrada de control está a 1, se permite que la información de las entradas *S* y *R* afecte al latch. El estado set se alcanza con $S = 1$, $R = 0$, y $C = 1$. Para cambiar al estado reset, las entradas deben ser $S = 0$, $R = 1$ y $C = 1$. En cualquier otro caso, cuando *C* retorna a 0, el circuito permanece en su estado actual. La entrada de control $C = 0$ desactiva el circuito para que el estado de las salidas no cambie, sin tener en cuenta los valores de *S* y *R*. Del mismo modo, cuando $C = 1$ y las entradas *S* y *R* son iguales a 0, el estado del circuito no cambia. Estas condiciones se muestran en la tabla de funcionamiento que acompaña al diagrama.

Cuando las tres entradas son iguales a 1 se produce un estado indefinido. Esta condición pone ambas entradas $\overline{S}\overline{R}$ del latch básico a 0, dando un estado indefinido. Cuando la entrada de control vuelve a ser 0, no se puede determinar el próximo estado, puesto que el latch ve las entradas (0, 0) seguidas por (1, 1). El latch *SR* con entrada de control es un circuito importante ya que otros latches y flip-flops se construyen en base a él. A veces se denomina flip-flop *RS* (o *SR*) al latch *SR* con entrada de control: sin embargo, según nuestra terminología, no está cualificado como flip-flop, puesto que el circuito no cumple los requisitos de los biestables presentados en la próxima sección.



(a) Diagrama lógico

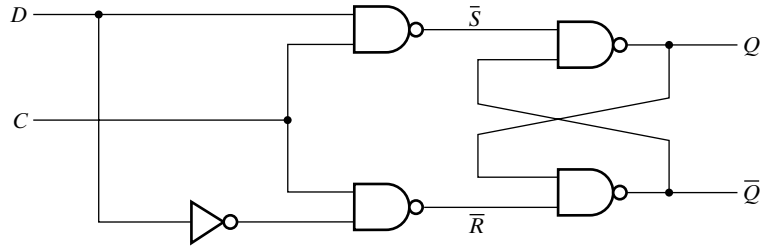
C	*S*	*R*	Siguiente estado de *Q*
0	***X***	***X***	No cambia
1	0	0	No cambia
1	0	1	$Q = 0$; estado «Reset»
1	1	0	$Q = 1$; estado «Set»
1	1	1	Indefinido

(b) Tabla de funcionamiento

□ **FIGURA 6-7**
Latch *SR* con entrada de control

Latch *D*

Una manera de eliminar el estado indefinido no deseable en el latch *SR* es asegurar que las entradas *S* y *R* nunca sean iguales a 1 al mismo tiempo. Esto se consigue con el latch *D* de la Figura 6-8. Este latch sólo tiene dos entradas: *D* (dato) y *C* (control). El complemento de la entrada *D* va directamente a la entrada $\overline{S}$, y se aplica *D* a la entrada $\overline{R}$. Si la entrada de control es 0, el latch tiene ambas entradas a nivel 1, y el circuito no puede cambiar de estado, con independencia del valor de *D*. La entrada de *D* se muestrea cuando $C = 1$. Si *D* es 1, la salida *Q* se



(a) Diagrama lógico

C	D	Siguiente estado de Q
0	X	No cambia
1	0	$Q = 0$; Estado «Reset»
1	1	$Q = 1$; Estado «Set»

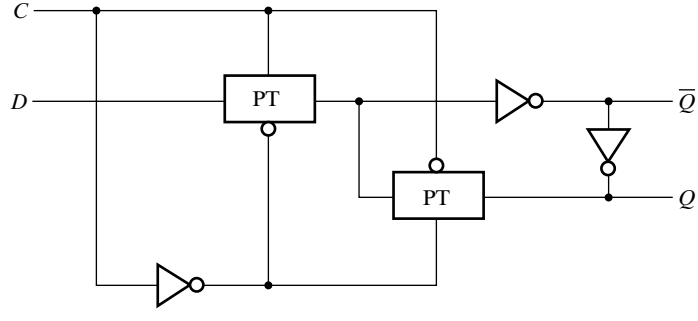
(b) Tabla de funcionamiento

□ **FIGURA 6-8**
Latch *D*

pone a 1, situando al circuito en el estado set. Si *D* es 0, la salida *Q* se pone a 0, llevando al circuito al estado reset.

El latch *D* recibe su nombre de su capacidad para retener el *dato* en su interior. La información binaria presente en la entrada de datos del latch *D* se transfiere a la salida *Q* cuando se habilita (1) la entrada de control. La salida sigue a los cambios en la entrada de datos, con tal de que la entrada de control esté habilitada. Cuando se deshabilita la entrada de control (0), la información binaria que estaba presente en la entrada de datos en el momento de la transición se retiene en la salida *Q* hasta que la entrada de control vuelva a habilitarse.

En los circuitos VLSI, el latch *D* se construye a menudo con puertas de transmisión (PTs), tal y como se muestra en la Figura 6-9. La PT se definió en la Figura 2-35. La entrada *C* controla dos PTs. Cuando $C = 1$, la PT conectada a la entrada *D* conduce, y la PT conectada a la salida *Q* está desconectada. Esto crea un camino desde la entrada *D*, a través de dos inversores, hasta la salida *Q*. Así, la salida sigue a la entrada de datos con tal de que *C* permanezca activo (1). Cuando *C* cambia a 0, la primera PT desconecta la entrada *D* del circuito, y la segundo PT conecta los dos inversores a la salida en un lazo. De este modo, el valor que estaba presente en la entrada *D* en el momento en que *C* pasó de 1 a 0 se retiene en la salida *Q* por el lazo.



□ **FIGURA 6-9**
Latch *D* con puertas de transmisión

6–3 FLIP-FLOPS

El estado de un latch en un flip-flop puede cambiar cuando hay un cambio momentáneo en el valor de la entrada de control. Este cambio se denomina *trigger* o *disparo,* y habilita el flip-flop. El latch *D* con pulsos de reloj en su entrada de control se dispara cada vez que aparece un pulso de nivel lógico l. Mientras que el pulso permanezca en el nivel activo (1), cualquier cambio en la entrada de datos cambiará el estado del latch. En este sentido, el latch es *transparente,* ya que su valor de entrada puede verse en las salidas.

Como muestra el diagrama de bloques de la Figura 6-3, un circuito secuencial tiene un camino de realimentación desde las salidas de los flip-flops hacia el circuito combinacional. Como consecuencia, las entradas de datos de los flip-flops se obtienen en parte de las salidas de los mismos y de otros flip-flops. Cuando los latches se emplean como elementos de almacenamiento aparece un serio problema. Las transiciones de estado de los latches empiezan en cuanto el pulso de reloj cambia al nivel lógico 1. Un nuevo estado aparece en la salida del latch mientras el pulso todavía esté activo. Esta salida está conectada a las entradas de algunos otros latches mediante un circuito combinacional. Si las entradas aplicadas a los latches cambian mientras el pulso de reloj todavía está en 1 lógico, los latches responderán a los *nuevos valores de estado* de los otros latches en lugar de a los *valores de estado originales,* y aparecerán una sucesión de cambios de estado en lugar de uno solo. El resultado es una situación imprevisible, puesto que el estado puede seguir cambiando y continuar cambiando hasta que el reloj vuelva al nivel lógico 0. El estado final dependerá de cuánto tiempo ha estado el pulso del reloj en el nivel lógico 1. A causa de este funcionamiento inestable, la salida de un latch no puede aplicarse ni directamente ni mediante lógica combinacional a la entrada del mismo u otro latch cuando todos los latches del sistema se disparan con una única señal de reloj.

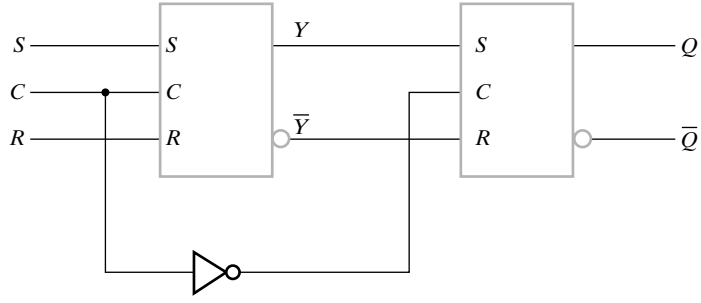
Los flip-flops se diseñan de manera que funcionen correctamente cuando son parte de un circuito secuencial con un solo reloj. Observe que el problema del latch es que es transparente: en cuanto una entrada cambia, un breve instante de tiempo después, la salida correspondiente responde cambiando. Esta transparencia es lo que permite que un cambio en la salida del latch produzca cambios adicionales en las salidas de otros latches mientras el pulso de reloj permanezca a 1. La clave del correcto funcionamiento de los flip-flops está en impedirles ser transparentes. En un flip-flop, antes de que una salida pueda cambiar, el camino de sus entradas hacia sus salidas ha de romperse. De este modo, un flip-flop no puede «ver» el cambio de sus salidas o de las salidas de otros flip-flops conectados en sus entradas, durante un mismo pulso de reloj. Así, el nuevo estado de un flip-flop sólo depende del estado inmediatamente anterior, y los flip-flops no pasan por múltiples cambios de estado.

Los latches pueden combinarse de dos maneras para formar un flip-flop. Una manera es combinar dos latches tal que (1) las entradas se presenten al flip-flop cuando haya un pulso de reloj en su estado de control y (2) el estado del flip-flop cambie sólo cuando no esté presente un pulso de reloj. Este circuito se denomina flip-flop *maestro-esclavo* (*master-slave*). Otra manera es crear un flip-flop que se dispare sólo durante *transición* de 0 a 1 (o de 1 a 0) en el reloj y que se deshabilite para los restantes instantes, incluyendo el pulso de reloj. Se dice de este circuito que es un flip-flop *disparado por flanco*. A continuación, se presentan aplicaciones de estos dos enfoques de disparo de flip-flops. Para la aproximación de disparo maestro-esclavo necesitaremos considerar el flip-flop SR mientras que un sencillo flip-flop D se comportará igual ante ambos tipos de disparos.

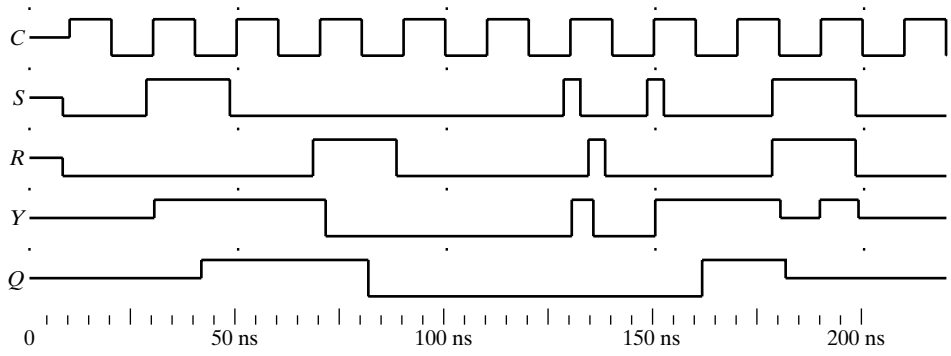
Flip-flop maestro-esclavo

En la Figura 6-10 se muestra un flip-flop *SR* maestro-esclavo formado por dos latches y un inversor. Los símbolos *S*, *C* y *R* son entradas de control para el latch *SR* (Figura 6-7) que se denomina como latch *SR* con entrada de reloj. En la Figura 6-10, el latch *SR* con entrada de reloj de la izquierda se denomina maestro, el derecho es el esclavo. Cuando la entrada de reloj *C* es 0, la salida del inversor es 1. Entonces, el esclavo se habilita, y su salida *Q* es igual a la salida principal *Y*. Mientras, el maestro está deshabilitado porque *C* es 0. Cuando se aplica un pulso de reloj con un nivel lógico 1, los valores en *S* y *R* controlan el valor *Y* almacenado en el maestro. El esclavo, sin embargo, está deshabilitado siempre que el pulso permanezca a 1, porque su entrada *C* es igual a 0. Cualquier cambio en las entradas externas *S* y *R* cambia la salida *Y* del maestro, pero sin afectar a la salida *Q* del esclavo. Cuando el pulso vuelve a ser 0, el maestro se deshabilita y se aísla de las entradas *S* y *R*. Al mismo tiempo, el esclavo se habilita, y el valor actual de *Y* se transfiere a la salida *Q* del flip-flop.

En la Figura 6-11 una simulación lógica ModelSim muestra cómo se comporta un flip-flop *SR* Maestro-Esclavo. Inicialmente, todos los valores son desconocidos incluso el reloj *C*. Cuando *S* y *R* valgan 0, y el reloj pase de 1 a 0, la salida *Y* del maestro y la salida *Q* del esclavo seguirán siendo desconocidas, puesto que de hecho, el valor anterior se ha almacenando. Cuando *S* está en 1 con *R* a 0 el flip-flop está preparado para responder con set en el próximo pulso de reloj. Cuando *C* se hace 1, *Y* se pone a 1. Cuando *C* retorna a 0, el valor *Y* del esclavo se copia en la salida *Q*, que se pone a 1. Después de que *S* vuelva a ser 0, *Y* y *Q* permanecen inalterados, almacenando el valor de 1 a lo largo del próximo periodo de reloj. Más tarde, *R* se vuelve 1. Tras la siguiente transición del pulso de reloj de 0 a 1, el maestro se pone 0 cambiando *Y* a 0. El esclavo no resulta afectado, porque su entrada *C* es 0. Puesto que el maestro es un circuito interior, su cambio de estado no está presente en la salida *Q*. Aun cuando las entradas *S* y *R* cambian durante este intervalo y el estado del maestro responde cambiando, las salidas del flip-flop permanecen en su anterior estado. Sólo cuando el pulso pasa a ser 0, se permite que la información del maestro pase a través del esclavo. En el ejemplo de la simulación, el valor $Y = 0$ se copia al esclavo haciendo que $Q = 0$. Observe que estos cambios tienen un retardo respecto de los cambios del pulso debido a los retardos de las puertas. Las entradas externas *S* y *R* también pueden cambiar en cualquier momento después de que el pulso de reloj haya pasado su transición negativa. Esto es porque, cuando la entrada *C* alcanza el 0, el maestro es deshabilitado, y *S* y *R* no tienen efecto hasta el próximo pulso de reloj. La siguiente sucesión de cambios en la señal muestra el comportamiento de «captura de unos» del flip-flop *SR* maestro-esclavo. Al comienzo de un pulso de reloj se presenta un estrecho pulso de nivel lógico 1 en la



□ **FIGURA 6-10**
Flip-flop maestro-esclavo *SR*



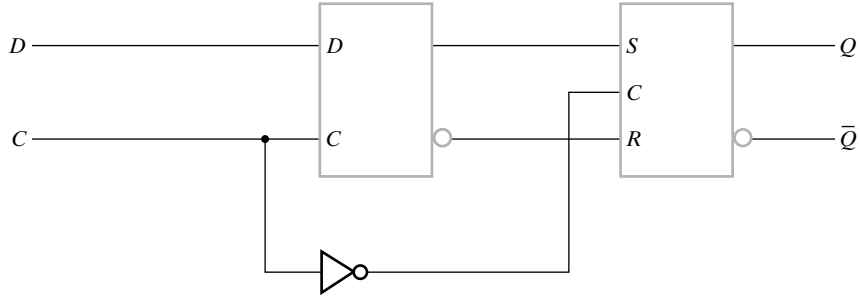
□ **FIGURA 6-11**
Simulación lógica de un flip-flop *SR* maestro-esclavo

entrada *S*. El maestro responde al 1 en *S* cambiando *Y* a 1. Entonces *S* vuelve a 0 mientras un estrecho pulso de nivel lógico 1 ocurre en *R*. El maestro responde al 1 en *R* volviendo atrás y cambiando *Y* a 0. Puesto que no hay más valores de nivel lógico 1 ni en *S* ni en *R*, el maestro continúa almacenando el último 0 que se copia al esclavo, cambiando *Q* a 0, en respuesta al cambio de transición del reloj a 0. Así, el maestro «capturó» ambos unos en *S* y en *R*. Ya que el último 1 se presentó en *R*, la salida *Q* ha permanecido a 0. En general, se supone que la «respuesta correcta» es la contestación a un valor de entrada cuando el reloj pasa a 0. Entonces, en este caso, la respuesta ha sido la correcta, aunque por accidente debido a los valores cambiantes en el maestro. Para el próximo pulso de reloj, un estrecho pulso de nivel lógico 1 se presenta en *S* que pone la salida *Y* del maestro a 1. El reloj pasa entonces a 0 y el valor 1 se transfiere al esclavo apareciendo en *Q*. En este caso, el valor correcto de *Q* debería ser 0 puesto que *Q* era 0 antes del pulso del reloj, y *S* y *R* eran 0 justo antes de que el reloj pasase a 0. El flip-flop está en un estado incorrecto con $Q = 1$, debido al éxito de la «captura de 1» en *S*. Para el último pulso del reloj, *S* y *R* se vuelven 1 antes de que el reloj pase a 0. Esto supone aplicar al maestro una combinación inválida que hace que las dos salidas *Y* e $\overline{Y}$ sean iguales a 1. Cuando el reloj cambia a 0, los biestables internos del maestro ven cómo sus entradas cambian de (0, 0) a (1, 1), llevando al latch a un estado desconocido que se transfiere inmediatamente a las entradas del esclavo que también entra en un estado desconocido. Esto demuestra que $S = 1$, $R = 1$ es una combinación de entrada no válida para el flip-flop *SR* maestro-esclavo.

Ahora considere un sistema secuencial que contiene muchos flip-flops del tipo maestro-esclavo, con las salidas de algunos conectadas a las entradas de otros. Suponga que el pulso del reloj de todos los flip-flops está sincronizado y se produce al mismo tiempo. Al principio de cada pulso del reloj, algunos de los maestros cambian de estado, pero todos los esclavos permanecen en sus estados anteriores. Esto significa que los biestables esclavos todavía permanecen en sus estados originales, mientras que los flip-flops maestros han cambiado hacia estados nuevos. Después de que los pulsos de reloj vuelvan a 0, algunos de los han cambiado de estado, pero ninguno de estos nuevos estados afectará a cualquiera de los siguientes maestros hasta el próximo pulso. De este modo, los estados de los flip-flops en un sistema síncrono pueden cambiar simultáneamente para un mismo pulso del reloj, aunque se conecten sus salidas a sus entradas o a las entradas de otros flip-flops. Esto es posible porque las entradas del flip-flop sólo afectan a su estado mientras el pulso del reloj es 1 y el estado nuevo sólo aparece en las salidas después de que el pulso del reloj ha vuelto a 0, asegurando así que los flip-flops no son transparentes.

Para el funcionamiento fiable del circuito secuencial, todas las señales deben propagarse desde las salidas de los flip-flops, a través del circuito combinacional, y realimentar las entradas del flip-flop maestro-esclavo, mientras el pulso de reloj permanece en el nivel lógico 0. Cualquier cambio que se produzca en las entradas del flip-flop después de que el pulso del reloj haya pasado al nivel lógico 1, intencionado o no, afectará al estado del flip-flop y puede producir el almacenamiento de valores incorrectos. Suponga que el retardo en el circuito combinacional es tal que *S* todavía está cambiando después de que el pulso del reloj haya pasado a 1. También suponga que, como consecuencia, el maestro se pone a 1 por la presencia de $S = 1$. Finalmente, cuando *S* deja de cambiar, está a 0, indicando que el estado del flip-flop no ha cambiado desde 0. Así, el valor 1 del maestro que se transferirá al esclavo es erróneo. De este comportamiento se derivan dos consecuencias. Primero, el flip-flop maestro-esclavo también se denomina flip-flop *disparado por pulso*, puesto que responde a los valores de entrada con un cambio de estado en cualquier instante durante su pulso del reloj. Segundo, el circuito debe diseñarse para que los retardos del circuito combinacional sean bastante cortos a fin de impedir que *S* y *R* cambien durante el pulso del reloj.

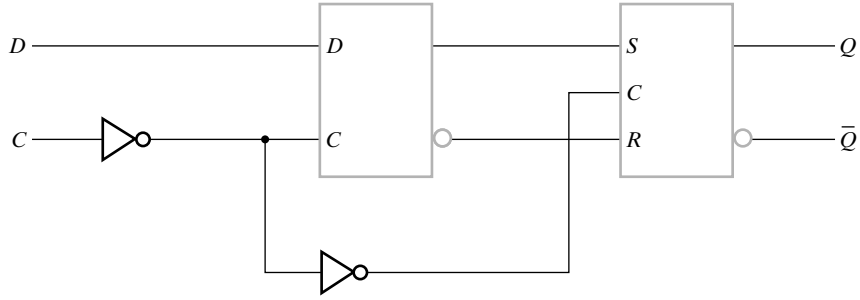
Un flip-flop *D* maestro-esclavo puede construirse a partir de un flip-flop *SR* maestro-esclavo sin más que reemplazar el latch *SR* maestro por un latch *D*. El circuito resultante se muestra en Figura 6-12. El circuito resultante cambia su valor en el flanco negativo del pulso de reloj tal y como lo hace el flip-flop *SR* maestro-esclavo. Sin embargo, el flip-flop tipo *D* no muestra un comportamiento típico de disparo por pulso. Más bien muestra un comportamiento *disparado por flanco*, y en este caso en concreto, un comportamiento activo por flanco negativo o de bajada. Así, un flip-flop *D* maestro-esclavo construido tal y como se ha indicado, también es un flip-flop activo por flanco o disparado por flanco.



□ **FIGURA 6-12**
Flip-flop *D* disparado por flanco negativo

Flip-flop disparados por flanco

Un flip-flop *disparado por flanco* ignora el pulso mientras está en un nivel constante y sólo se dispara durante una transición de la señal de reloj. Algunos flip-flops disparados por flanco se activan en el flanco positivo (transición de 0 a 1), mientras que otros se activan en el flanco negativo (transición de 1 a 0) como se ha ilustrado en la subsección anterior. El diagrama lógico de un flip-flop tipo *D* disparado por flanco positivo que será analizado aquí en detalle aparece en la Figura 6-13. Este flip-flop toma exactamente la forma de un flip-flop maestro-esclavo, siendo el maestro un latch *D* y el esclavo un latch *SR* o un latch *D*. También, se añade un inversor a la entrada del reloj. Debido a que el maestro es un latch *D*, los flip-flops muestran un comportamiento activo por flanco en lugar de maestro-esclavo o activo por pulso. Cuando la



□ **FIGURA 6-13**
Flip-flop *D* disparado por flanco positivo

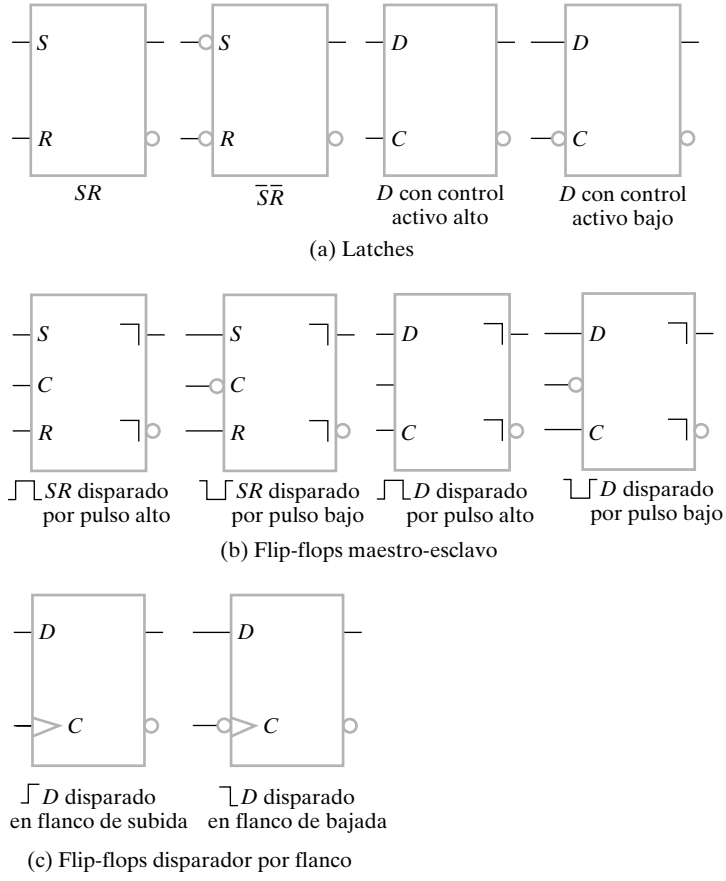
entrada de reloj es igual a 0, el maestro se habilita y se hace transparente de modo que *D* sigue al valor de la entrada. El esclavo es inhabilitado y mantiene el estado anterior del flip-flop fijo. Cuando se produce un flanco positivo, la entrada de reloj cambia a 1. Esto desactiva al maestro fijando su valor y habilita al esclavo para que copie el estado del maestro en la salida. El estado del maestro que se copia en la salida es el estado presente en el flanco positivo del reloj. De esta manera, el comportamiento parece ser disparado por flanco. Cuando la entrada de reloj es igual a 1, el maestro es deshabilitado y no puede cambiar, de modo que el estado del maestro y del esclavo permanezcan inalterados. Finalmente, cuando el reloj cambia de 1 a 0, el maestro se habilita y comienza siguiendo al valor de *D*. Pero durante la transición de 1 a 0, el esclavo se deshabilita antes de que pueda alcanzar cualquier cambio del maestro. Así, el valor almacenado en el esclavo permanece inalterado durante esta transición. Al final del capítulo, en el Problema 6-3 se da una implementación alternativa.

Símbolos gráficos estándar

En la Figura 6-14 se muestran los símbolos gráficos estándar para los diferentes tipos de latches y flip-flops. Los flip-flops y los latches se designan por un bloque rectangular con las entradas a la izquierda y las salidas a la derecha. Una de las salidas designa el estado normal del flip-flop, y la otra, con un círculo, designa la salida complementada. El símbolo gráfico para el latch *SR* o el flip-flop *SR* tiene las entradas *S* y *R* indicadas dentro del bloque. En el caso del latch $\overline{S}\overline{R}$, se añaden unos círculos a las entradas para indicar que el set y el reset son entradas activas por nivel lógico 0. El símbolo para el latch *D* o el flip-flop *D* tiene las entradas *D* y *C* indicadas dentro del bloque.

Debajo de cada símbolo, aparece un título descriptivo que no es parte del símbolo. En los títulos, ⎍ denota un pulso positivo, ⎏ un pulso negativo, ⌜ un flanco positivo y ⌞ un flanco negativo.

En los símbolos de los latches, el disparo por nivel 0 en lugar de por nivel 1 se denota añadiendo un círculo a la entrada de activación. El maestro-esclavo es un flip-flop disparado por pulso y se indica como tal con un símbolo en ángulo recto denominado *indicador de salida pospuesto* delante de las salidas. Este símbolo muestra que la señal de salida cambia al final del pulso. Para denotar que el maestro-esclavo responde a un pulso negativo (es decir, un pulso a 0 con el valor del reloj inactivo a 1), se pone un círculo en la entrada de *C*. Para denotar que el flip-flop disparado por flanco responde a un flanco, se coloca el símbolo de una punta de flecha delante de la letra *C* designando una *entrada dinámica*. Este símbolo de indicador dinámico denota el hecho de que el flip-flop responde a transiciones de flancos en los pulsos de reloj de



□ **FIGURA 6-14**
Símbolos gráficos estándar para latches y flip-flops

entrada. Un círculo fuera del bloque adyacente al indicador dinámico indica que el circuito se activa con una transición de flanco negativa. La ausencia de un círculo designa una activación o disparo por transición de flanco positiva.

A menudo, todos los flip-flops empleados en un circuito son del mismo tipo de disparo, como activos por flanco positivo. Entonces, todos los flip-flops cambiarán respecto al mismo evento de reloj. Al usar flip-flops con diferentes tipos de disparo en un mismo circuito secuencial, uno puede desear que todas las salidas de los biestables cambien en relación al mismo evento. Esos flip-flops que se comportan de una manera contraria a la transición de polaridad adoptada pueden cambiarse añadiendo inversores a sus entradas de reloj. Un procedimiento preferido es proporcionar los pulsos positivos y negativos desde el generador de reloj principal cuidadosamente alineados. Nosotros aplicamos los pulsos positivos a los flip-flops disparados por pulso positivo (maestro-esclavo) y a los flip-flops disparados por flanco negativo y aplicamos los pulsos negativos a los flip-flops activos por pulso negativo (maestro-esclavo) y a los flip-flops activos por flanco positivo. De este modo, todas las salidas de los flip-flops cambiarán al mismo tiempo. Finalmente, para prevenir problemas concretos de sincronización, algunos diseñadores usan flip-flops con diferentes disparos (es decir, flip-flops disparados por flanco positivo y flip-flops disparados por flanco negativo) con un solo reloj. En estos casos, se provoca intencionadamente que las salidas de los biestables cambien en momentos diferentes.

En este texto, suponemos que todos los flip-flop son disparados por flanco positivo, a menos que se indique lo contrario. Esto mantiene un símbolo gráfico uniforme para los flip-flops y cronogramas consistentes.

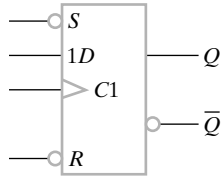
Véase que no hay ninguna entrada al flip-flop *D* que produzca una condición de «no cambio». Esta condición puede lograrse desactivando los pulsos de reloj en la entrada *C* o dejando los pulsos de reloj y conectando la salida de modo que realimente la entrada *D* a través de un multiplexor cuando el estado del flip-flop deba permanecer igual. La técnica que desactiva los pulsos del reloj se denomina *gating* del reloj. Esta técnica normalmente necesita menos puertas y disminuye la potencia consumida, pero se evita a menudo porque los pulsos del reloj, al atravesar las puertas, llegan a los flip-flops con retardos. El retardo, denominado *skew* provoca que los flip-flops con reloj modificado con puertas y los que acceden directamente al reloj cambien en momentos diferentes. Esto puede convertir el circuito en inestable, debido a que las salidas de algunos flip-flops pueden alcanzar a otros mientras sus entradas todavía afectan a su estado.

Entradas asíncronas

A menudo, los flip-flop proporcionan entradas especiales asíncronas de set y reset (es decir, independientes de la entrada de reloj *C*). Las entradas que asincrónicamente ponen el flip-flop en set se denominan *set asíncrono* o *preset*. Se llaman *reset asíncrono* o *clear* las entradas que asíncronamente resetean el flip-flop. La aplicación de un 1 lógico (o un 0 lógico si hay un círculo) en estas entradas afecta a la salida del flip-flop en ausencia de señal de reloj. Cuando se conecta un sistema digital, los estados de sus flip-flops pueden ser cualquiera. Las entradas directas son útiles para colocar, a priori, los flip-flops de un sistema digital en un estado inicial normal para el funcionamiento con reloj.

En la Figura 6-15(a) se muestra el símbolo gráfico estándard IEEE para un flip-flop *D* disparado por flanco positivo con entradas directas de set y reset. Las anotaciones, *C*1 y 1*D*, ilustran la dependencia del control. Una entrada nombrada como C_n donde *n* es cualquier número, controla las otras entradas que empiezan con el número *n*. En la figura, *C*1 controla la entrada 1*D*. *S* y *R* no tienen ningún 1 delante, y por consiguiente, no están controladas por la entrada de reloj *C*1. Las entradas *S* y *R* tienen círculos en sus líneas de entrada indicando que son activas a nivel bajo (es decir, un 0 aplicado producirá la acción set o reset).

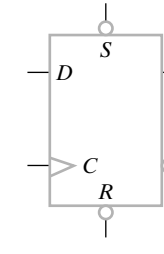
La tabla de la función en la Figura 6-15(b) especifica el funcionamiento del circuito. Las primeras tres filas de la tabla especifican el funcionamiento de las entradas asíncronas *S* y *R*. Estas entradas se comportan como las entradas del latch $\overline{S}\overline{R}$ NAND (*véase* la Figura 6-6), ope-



(a) Símbolo gráfico

S	*R*	*C*	*D*	*Q*	$\overline{Q}$
0	1	*X*	*X*	1	0
1	0	*X*	*X*	0	1
0	0	*X*	*X*	Indefinido	
1	1	↑	0	0	1
1	1	↑	1	1	0

(b) Tabla de funcionamiento



(c) Símbolo simplificado

□ **FIGURA 6-15**
Flip-flop *D* con entradas asíncronas de set y reset

rando independientemente del reloj. Las últimas dos filas de la tabla de funcionamiento especifican el funcionamiento síncrono para los valores de *D*. El reloj *C* se muestra con una flecha ascendente para indicar que el flip-flop es disparado por flanco positivo. Los efectos sobre la entrada *D* son controlados por el reloj de la manera habitual.

La Figura 6-15(c) muestra un símbolo menos formal para el flip-flop disparado por flanco positivo con entradas asíncronas de set y reset. La posición de *S* y *R* arriba y abajo del símbolo en lugar de a la izquierda del borde implica que esos cambios de salida resultantes no se controlan por el reloj *C*.

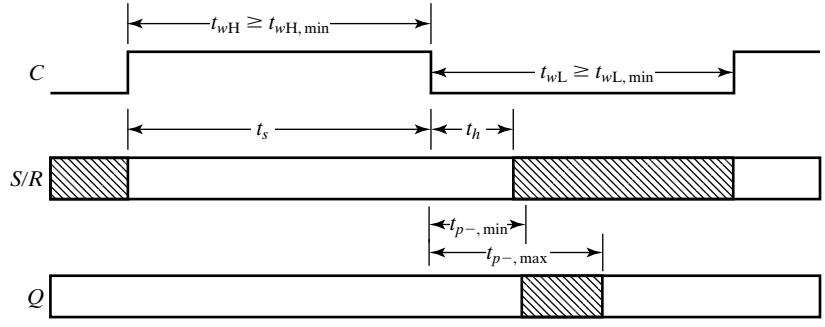
Tiempos de los flip-flops

Hay parámetros de tiempo asociados con el funcionamiento de los dos flip-flops disparados por pulso y activos por flanco. Estos parámetros se ilustran para un flip-flop *SR* maestro-esclavo y para un flip-flop *D* disparado por flanco negativo en la Figura 6-16. Los parámetros para el flip-flop *D* disparado por flanco positivo son los mismos excepto que son referenciados al flanco positivo del reloj en lugar de al flanco negativo del reloj.

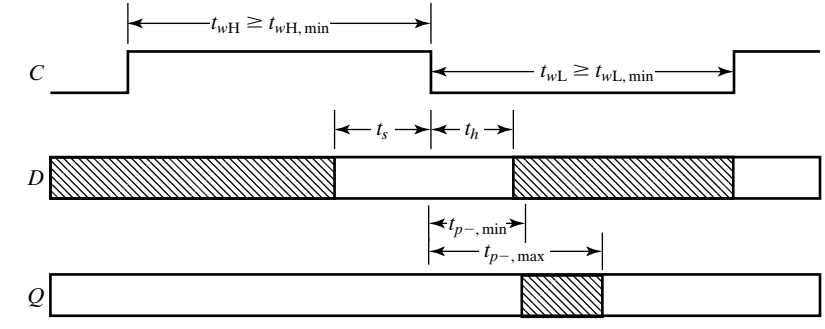
Al emplear los flip-flops, debe tenerse en cuenta el tiempo de respuesta a sus entradas y a la entrada de reloj *C*. Para ambos flip-flops, hay un tiempo mínimo llamado *tiempo de setup*, t_s, es el tiempo anterior a la ocurrencia de la transición del reloj que provoca un cambio en la salida durante el cual las entradas *S*, *R* o *D* deben mantener un valor constante. De no ser así, en el caso de un flip-flop maestro-esclavo, el maestro podría cambiarse erróneamente o podría adquirir un valor intermedio en el momento en que lo copiase el esclavo si es un flip-flop disparado por flanco. De manera similar hay un tiempo mínimo denominado *tiempo de mantenimiento* (*tiempo de hold*), t_h es el tiempo posterior a la transición del reloj que causa la salida, durante el cual las entradas *S*, *R* o *D* no deben cambiar. Si esto no fuera así, el maestro podría responder al cambio de la entrada y podría estar cambiando en el momento en el que esclavo lo copia. Además hay un ancho mínimo para el pulso de reloj t_w que asegura que el maestro tiene tiempo suficiente para capturar el valor de entrada correctamente. Entre estos parámetros, la mayoría de los flip-flop disparados por pulsos y activos por flanco difieren en el tiempo de setup tal y como muestra la Figura 6-16. El flip-flop disparado por pulso tiene un tiempo de setup igual a la anchura de pulso de reloj, considerando que el tiempo del setup para el flip-flop disparado por flanco puede ser mucho menor que la anchura de pulso de reloj. Como consecuencia, los flip-flops activados por flanco tienden a proporcionar los diseños más rápidos puesto que las entradas pueden cambiar más tarde con respecto al próximo flanco del reloj activo.

Se definen los *tiempos de retardo de propagación*, t_{PHL}, t_{PLH} o t_{pd} de los flip-flops como el intervalo entre el flanco activo del reloj y la estabilización de la salida hacia un nuevo valor. Estos tiempos se definen del mismo modo que los tiempos de retardo para un inversor sólo que los valores son medidos desde el flanco del reloj activo en lugar de desde la entrada del inversor. En Figura 6-16, todos estos parámetros se designan por t_{P-} y se dan los valores mínimos y máximos. Puesto que los cambios de las salidas de los flip-flops van aparte de las entradas de control, el tiempo de retardo de propagación mínimo debe ser más largo que el tiempo de mantenimiento para el funcionamiento correcto. Éstos y otros parámetros se especifican en los catálogos que los fabricantes ofrecen para sus productos.

Parámetros temporales similares se pueden definir para los latches y para las entradas asíncronas, con retardos de propagación adicionales necesarios para modelar el comportamiento transparente de los latches.



(a) Disparado por pulso (alto)



(b) Disparado por flanco (de bajada)

□ **FIGURA 6-16**
Parámetros de tiempo de un flip-flop

6-4 ANÁLISIS DE CIRCUITOS SECUENCIALES

El comportamiento de un circuito secuencial viene determinado por las entradas, salidas, y por el estado actual del circuito. Las salidas y el estado futuro son función de las entradas y del estado actual. El análisis de un circuito secuencial consiste en obtener una descripción conveniente que demuestre la sucesión en el tiempo de entradas, salidas y estados.

Un diagrama lógico se reconoce como un circuito secuencial síncrono si incluye los flip-flop con las entradas del reloj conectadas directamente o indirectamente a una señal de reloj y si las entradas directas de set y reset permanecen sin usar durante el funcionamiento normal del circuito. Los flip-flops pueden ser de cualquier tipo, y el diagrama lógico puede o no puede incluir puertas combinacionales. En esta sección, se muestra una representación algebraica para especificar el diagrama lógico de un circuito secuencial. Se presentan una tabla de estado y el diagrama de estado que describen el comportamiento del circuito. Se emplearán ejemplos específicos a lo largo de la discusión para ilustrar los diferentes procedimientos.

Ecuaciones de entrada

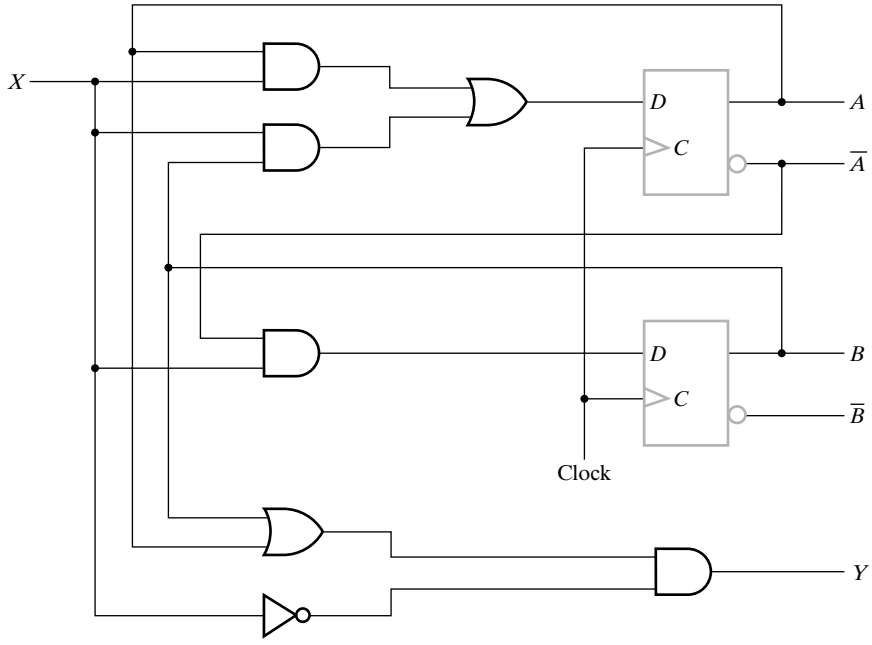
El diagrama lógico de un circuito secuencial consiste en flip-flops y, normalmente, en puertas combinacionales. Para dibujar el diagrama lógico del circuito secuencial toda la información que se necesita es conocer el tipo de flip-flops empleados y una lista de funciones booleanas

para el circuito combinacional. La parte del circuito combinacional que genera las señales para las entradas de los flip-flops puede describirse por un juego de funciones booleanas denominado *ecuaciones de entrada a los flip-flops*. Adoptaremos la convención de emplear el símbolo de la entrada al flip-flop para nombrar la variable de la ecuación de entrada a dicho flip-flop y usaremos el nombre de la salida del flip-flop como el subíndice para la variable. A partir de este ejemplo, es claro que la ecuación de entrada al flip-flop es la expresión booleana para un circuito combinacional. El símbolo con subíndice denota una variable de salida del circuito combinacional. Esta salida siempre se conecta a la entrada del flip-flop de ahí el nombre «ecuación de entrada al flip-flop».

Las ecuaciones de entrada de los flip-flops constituyen una expresión algebraica conveniente para especificar el diagrama lógico de un circuito secuencial. Suponen el tipo de flip-flop a partir del símbolo de la letra, y especifican totalmente el circuito combinacional que maneja los flip-flop. El tiempo no está explícitamente incluido en estas ecuaciones, pero está implícito en el reloj y en la entrada *C* de los flip-flops. En la Figura 6-17 se muestra un ejemplo de un circuito secuencial. El circuito tiene dos flip-flops tipo *D*, una entrada *X*, y una salida *Y* que puede especificarse por las siguientes ecuaciones:

$$D_A = AX + BX$$
$$D_B = \bar{A}X$$
$$Y = (A + B)\bar{X}$$

Las primeras dos ecuaciones son para las entradas del flip-flop, y la tercera ecuación especifica la salida *Y*. Observe que las ecuaciones de las entradas emplean el símbolo *D* que es el mismo que el símbolo de la entrada de los flip-flops. Los subíndices *A* y *B* designan las respectivas salidas de los flip-flops.



□ **FIGURA 6-17**
Ejemplo de un circuito secuencial

Tabla de estados

Pueden enumerarse las relaciones funcionales entre las entradas, salidas, y los estados de los flip-flops de un circuito secuencial en una tabla de estados (o transiciones). La tabla de estados para el circuito de la Figura 6-17 se muestra en la Tabla 6-1. La tabla está formada por cuatro secciones, etiquetadas como *estado actual*, *entradas*, *estado futuro*, y *salida*. La sección del estado actual muestra los estados de los flip-flops *A* y *B* en cualquier instante de tiempo *t* dado. La sección de la entrada da cada valor de *X* para cada posible estado actual. Observe que para cada posible combinación de la entrada, cada uno de los estados actuales aparece repetido. La sección del estado futuro muestra el estado de los flip-flops un periodo de reloj más tarde, en el momento $t + 1$. La sección de salida da el valor *Y* para un instante de tiempo *t* para cada combinación de estado actual y entrada.

□ **TABLA 6-1**
Tabla de estados para el circuito de la Figura 6-17

Estado actual		Entrada	Estado futuro		Salida
A	***B***	***X***	***A***	***B***	***Y***
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Obtener una tabla de estado consiste primero en enumerar todas las posibles combinaciones binarias de estados actuales y entradas. En la Tabla 6-1, hay ocho combinaciones binarias, desde 000 hasta 111. Entonces, se determinan los valores de los estados futuros a partir del diagrama lógico o de las ecuaciones de entrada de los flip-flops. Para un flip-flop *D*, se mantiene la relación $A(t + 1) = D_A(t)$. Esto significa que el estado futuro del flip-flop *A* es igual al valor actual en su entrada *D*. El valor de la entrada *D* está especificado en la ecuación de entrada del flip-flop como una función del estado actual de *A* y *B* y una entrada *X*. Además, el estado futuro del flip-flop *A* debe satisfacer la ecuación

$$A(t + 1) = D_A = AX + BX$$

La sección del estado futuro en la tabla de estado, debajo de la columna *A*, tiene tres 1 donde el estado actual y el valor de la entrada satisfacen las condiciones $(A, X) = 11$ o $(B, X) = 11$. De manera similar, el estado futuro del flip-flop *B* se obtiene de la ecuación de entrada

$$B(t + 1) = D_B = \bar{A}X$$

y es igual a 1 cuando el estado actual de *A* es 0 y la entrada *X* es igual a 1. La columna de salida se obtiene de la ecuación de salida

$$Y = A\bar{X} + B\bar{X}$$

La tabla de estado de cualquier circuito secuencial con flip-flops del tipo *D* se obtiene de esta manera. En general, un circuito secuencial con *m* flip-flops y *n* entradas necesita 2^{m+n} filas en la tabla de estado. Los números binarios desde 0 y hasta $2^{m+n} - 1$ se listan combinando las columnas de entrada y de estado actual. La sección del estado futuro tiene *m* columnas, una para cada flip-flop. Los valores binarios para el estado futuro se obtienen directamente de las ecuaciones de entrada de cada flip-flop *D*. La sección de salida tiene tantas columnas como variables de salida. Sus valores binarios se obtienen del circuito o de las funciones booleanas de la misma manera que en una tabla de verdad.

La Tabla 6-1 es unidimensional en el sentido de que el estado actual y las combinaciones de entrada se combinan en una única columna. También se emplea frecuentemente una tabla de estado bidimensional en la que el estado actual se coloca en la columna de la izquierda y las entradas en la fila superior. El estado futuro se coloca en cada celda de la tabla para la combinación correspondiente del estado actual y de la entrada. Una tabla bidimensional similar se emplea para las salidas si dependen de las entradas. Esta tabla de estado se muestra en la Tabla 6-2. Los circuitos secuenciales en los que las salidas dependen de las entradas, así como de los estados presentes, se denominan *Autómatas o Máquinas de Mealy*. En cambio, si las salidas sólo dependen de los estados actuales, entonces basta con una única columna unidimensional. En este caso, los circuitos se denominan *Autómatas de Moore*. Cada modelo se nombra por su creador.

□ **TABLA 6-2**
Tabla de estados bidimensional para el circuito de la Figura 6-17

Estado actual		**Estado futuro**				**Salida**	
		***X* = 0**		***X* = 1**		***X* = 0**	***X* = 1**
A	***B***	***A***	***B***	***A***	***B***	***Y***	***Y***
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

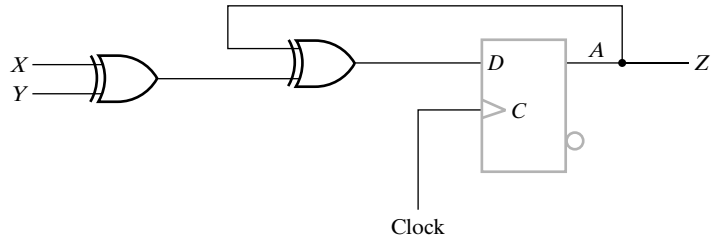
Como ejemplo de un Autómata de Moore, suponga que queremos obtener el diagrama lógico y la tabla de estado de un circuito secuencial especificado por la siguiente ecuación de entrada al flip-flop:

$$D_A = A \oplus X \oplus Y$$

y por la ecuación de salida:

$$Z = A$$

El símbolo D_A implica un flip-flop tipo *D* con la salida designada por la letra *A*. Las variables *X* y *Y* se toman como las entradas y *Z* como la salida. En la Figura 6-18 se muestran el diagrama lógico y la tabla de estados para este circuito. La tabla de estados tiene una columna para el estado actual y una columna para las entradas. El estado futuro y la salida también están en columnas simples. El estado futuro se obtiene de la ecuación de la entrada del flip-flop que determina una función de paridad impar. (*Véase* la Sección 2-8.) La columna de la salida simplemente es una copia de la columna para el estado actual *A*.



(a)

Estado actual	Entradas		Estado futuro	Salida
A	*X*	*Y*	*A*	*Z*
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

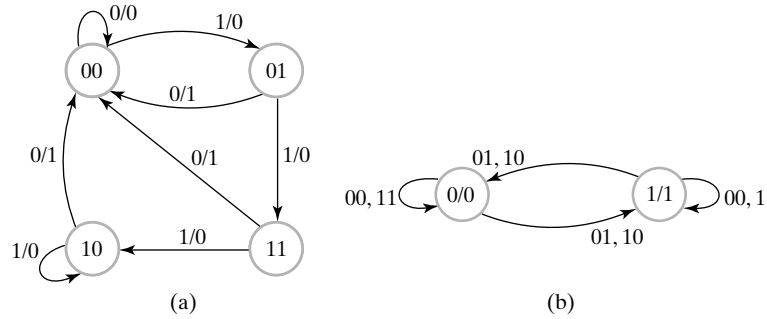
(b) Tabla de estados

□ **FIGURA 6-18**
Diagrama lógico y tabla de estados para $D_A = A \oplus X \oplus Y$

Diagrama de estados

La información disponible en una tabla de estados se puede representar gráficamente en forma de diagrama de estados. En este tipo de diagrama, un estado se representa por un círculo, y las transiciones entre los estados se indican mediante líneas orientadas que conectan los círculos. En la Figura 6-19 se dan ejemplos de diagramas de estados. La Figura 6-19(a) muestra el diagrama de estados para el circuito secuencial de la Figura 6-17 y su tabla de estados en la Tabla 6-1. El diagrama de estados proporciona la misma información que la tabla de estados y se obtiene directamente de él. El número binario dentro de cada círculo identifica el estado de los flip-flops. En los Autómatas de Mealy, las líneas orientadas se etiquetan con dos números binarios separados por una barra /. El valor de la entrada durante el estado actual precede a la barra, y el siguiente valor tras la barra es el valor de la salida durante el estado actual aplicando dicha entrada. Por ejemplo, la línea orientada que va del estado 00 al 01 se etiqueta 1/0, significando que cuando el circuito secuencial está en el estado actual 00 y la entrada es 1, la salida es 0. Después de la próxima transición del reloj, el circuito va al siguiente estado, 01. Si la entrada cambia a 0, entonces la salida se vuelve 1, pero si la entrada permanece a 1, la salida permanece a 0. Esta información se obtiene del diagrama de estados a lo largo de las dos líneas orientadas que parten del círculo con estado 01. Una línea orientada que conecta un círculo consigo mismo indica que no se produce ningún cambio de estados.

El diagrama de estados de la Figura 6-19(b) es para el circuito secuencial de la Figura 6-18. Aquí, sólo se necesita un único flip-flop con dos estados. Hay dos entradas binarias, y la salida sólo depende del estado del flip-flop. En el Autómata de Moore, la barra no aparece en las



□ **FIGURA 6-19**
Diagramas de estados

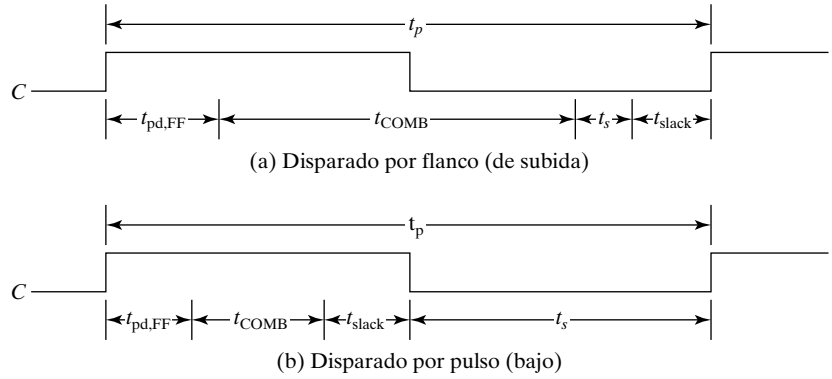
líneas orientadas puesto que las salidas sólo dependen del estado y no de los valores de la entrada. En cambio, dentro del círculo y separada del estado por una barra, sí se incluye la salida. En el diagrama, por cada transición de estado hay dos condiciones de entrada separadas por una coma. Cuando hay dos variables de entrada, cada estado puede tener hasta cuatro líneas orientadas partiendo del círculo correspondiente, dependiendo del número de estados y del próximo estado para cada combinación binaria de los valores de entrada.

No hay ninguna diferencia entre una tabla de estados y un diagrama de estados, salvo su manera de representarse. La tabla de estados se obtiene fácilmente a partir de un diagrama lógico dado y de las ecuaciones de la entrada. El diagrama de estados sigue directamente a la tabla de estados. El diagrama de estados muestra una representación gráfica de las transiciones de estados y es la forma más apropiada para la interpretación humana del funcionamiento del circuito. Por ejemplo, el diagrama de estados de la Figura 6-19(a) muestra claramente que, empezando en el estado 00, la salida es 0 con tal de que la entrada permanezca a 1. La primera entrada a 0 después de una serie de 1 da una salida de 1 y envía el circuito hacia atrás, al estado inicial 00. El diagrama de estados de la Figura 6-19(b) muestra cómo el circuito permanece en un estado dado con tal de que las dos entradas tengan el mismo valor (00 o 11). Sólo hay una transición de estados entre los dos posibles estados cuando ambas entradas son diferentes (01 o 10).

Temporización del circuito secuencial

Además de analizar la función de un circuito, también es importante analizar su comportamiento en términos de *máximo retardo de la entrada hacia la salida* y *máxima frecuencia de reloj*, f_{max} a la que puede funcionar. En primer lugar, la frecuencia de reloj es simplemente la inversa del periodo de reloj t_p mostrado en la Figura 6-20. Así que, la máxima frecuencia de reloj permitida corresponde al mínimo periodo de reloj aceptable, t_p. Para determinar el valor mínimo permitido para el periodo de reloj, necesitamos determinar el mayor retardo desde el flanco de disparo del reloj hasta el próximo flanco de disparo de reloj. Estos retardos se miden en todos los caminos posibles del circuito bajo los cuales se propagan las señales que cambian.

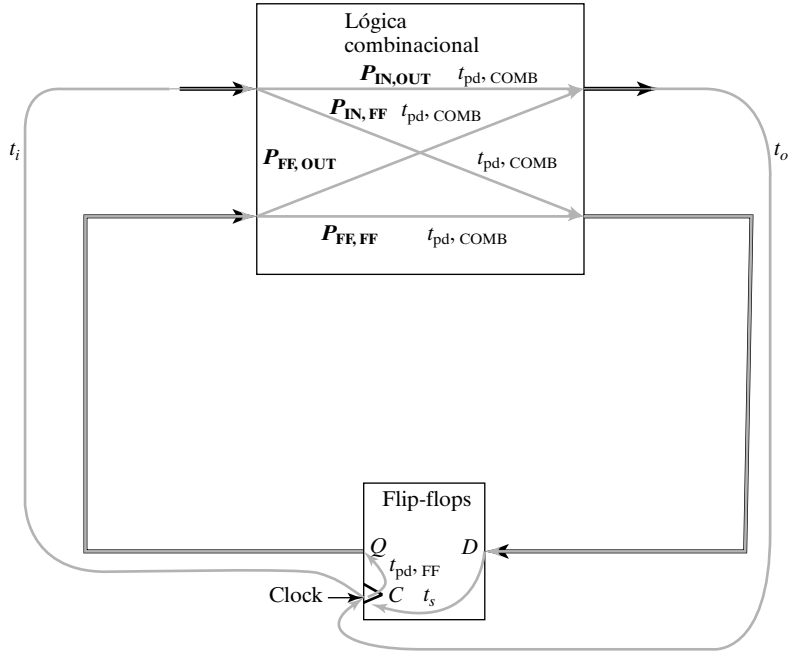
Cada uno de éstos retardos tiene tres componentes: (1) un retardo de propagación del flip-flop, $t_{pd,\,FF}$, (2) un retardo de la lógica combinacional a través de la serie de puertas a lo largo del camino, $t_{pd,\,COMB}$, y (3) un tiempo setup del flip-flop, t_s. Cuando un cambio en la señal se propaga a lo largo del camino, se retrasa sucesivamente una cantidad igual a cada uno de estos retardos. Observe que hemos empleado t_{pd}, en lugar de valores más concretos, t_{PLH} y t_{PHL}, tanto



□ **FIGURA 6-20**
Parámetros temporales del circuito secuencial

para los flip-flops como para las puertas lógicas combinacionales para simplificar los cálculos del retardo. La Figura 6-20 resume el gráfico del retardo para los flip-flops activos por flanco y activos por pulso.

Después de un flanco positivo de reloj, si un flip-flop cambia, su salida cambia un tiempo $t_{p,\,FF}$ después del flanco de reloj. Este cambio entra en la lógica combinacional y se propaga hacia la entrada de un flip-flop. Se necesita un tiempo adicional, $t_{pd,\,COMB}$, para que dicho cambio alcance al segundo flip-flop. Finalmente, antes del próximo flanco positivo del reloj, este cambio debe mantenerse en la entrada del flip-flop durante un tiempo de setup t_s. Este camino, $C_{FF,\,FF}$ se muestra junto con otros caminos posibles en la Figura 6-21. En los caminos $C_{IN,\,FF}$ conectados a las entradas del primero, el $t_{pd,\,FF}$ se reemplaza por t_i que es el último tiempo que



□ **FIGURA 6-21**
Caminos temporales para circuitos secuenciales

la entrada cambia después del flanco del reloj positivo. Para un camino $C_{FF, OUT}$ conectado a las salidas primarias, el tiempo t_s se reemplazan por t_o que es el último tiempo que la salida se permite cambiar antes del próximo flanco del reloj. Finalmente, en el Modelo de Mealy, pueden aparecer caminos combinacionales desde la entrada a la salida, $C_{IN, OUT}$ que emplean tanto t_i como t_o. Cada camino tiene un tiempo t_{slack}, es un tiempo extra permitido más allá del periodo de reloj necesario para el camino. De la Figura 6-21, resulta la siguiente ecuación para un camino de tipo $C_{FF, FF}$:

$$t_p = t_{slack} + (t_{pd, FF} + t_{pd, COMB} + t_s)$$

para asegurar que cualquier cambio de valor es capturado por el flip-flop receptor, los t_{slack} deben ser mayores o iguales a cero para todos los caminos. Esto requiere que

$$t_p \geqslant \max (t_{pd, FF} + t_{COMB} + t_s) = t_{p, min}$$

donde el máximo se toma sobre todos los caminos en los que se propagan las señales de flip-flop en flip-flop. El próximo ejemplo presenta los cálculos representativos para los caminos $C_{FF, FF}$.

EJEMPLO 6-1 Cálculo del periodo y la frecuencia del reloj

Suponga que todos los flip-flops empleados son del mismo tipo y que tienen $t_{pd} = 0.2$ ns (nanosegundo $= 10^{-9}$ segundos) y $t_s = 0.1$ ns. Entonces el camino más largo empezando y acabando con un flip-flop será el camino con el $t_{pd, COMB}$ más grande. Es más, suponga que el mayor $t_{pd, COMB}$ es 1.3 ns y que t_p se ha fijado a 1.5 ns. De la ecuación anterior para t_p, podemos escribir

$$1.5 \text{ ns} = t_{slack} + 0.2 + 1.3 + 0.1 = t_{slack} + 1.6 \text{ ns}$$

Resolviendo, tenemos $t_{slack} = -0.1$ ns, con lo que este valor de t_p es demasiado pequeño. A fin de que t_{slack} sea mayor o igual a cero para el camino más largo, $t_p \geqslant t_{p, min} = 1.6$ ns. La máxima frecuencia $f_{max} = 1/1.6$ ns $= 625$ MHz (megahertzio $= 10^6$ ciclos por segundo). Observamos que, si t_p es demasiado grande para satisfacer las especificaciones del circuito, o empleamos células lógicas más rápidas o debemos cambiar el diseño del circuito para acortar los caminos problemáticos del circuito manteniendo siempre la función deseada. ■

Es interesante apreciar cómo el tiempo de mantenimiento del flip-flop no aparece en la ecuación del periodo de reloj. Se relaciona con otra ecuación de tiempos que trata con una o dos situaciones específicas. En un caso, los cambios de salida llegan demasiado pronto a las entradas de uno o más flip-flops. En el otro caso, las señales de reloj que alcanzan uno o más flip-flops se retardan de algún modo, una condición denominada *skew* del reloj. El skew del reloj también puede afectar a la frecuencia del reloj máxima.

Simulación

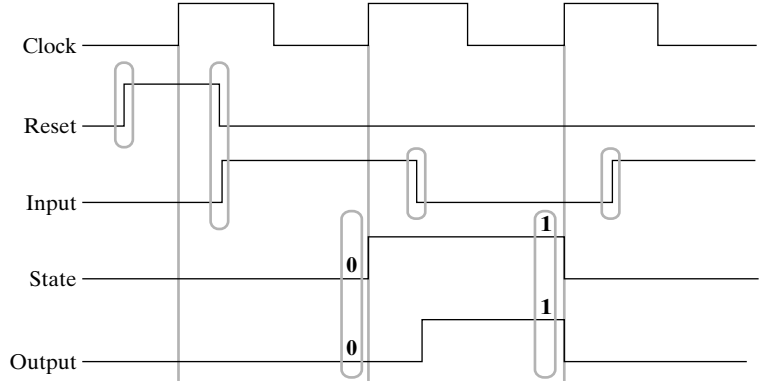
La simulación del circuito secuencial implica problemas que no se presentan en los circuitos combinacionales. En primer lugar, en lugar de un conjunto de patrones de entrada para los que el orden de aplicación es irrelevante, hay que aplicar los patrones como una secuencia. Esta secuencia supone la oportuna aplicación de los patrones y de pulsos del reloj. Segundo, debe existir alguna forma de colocar al circuito en un estado conocido. En realidad, la inicialización a

un estado conocido se lleva a cabo al principio de la simulación aplicando una subsecuencia de inicialización. En el caso más simple, esta subsecuencia es una señal de reset. Para flip-flops que carecen de reset (o set), se necesita una sucesión más larga que consiste normalmente en un reset inicial seguido por una sucesión de patrones normales de entrada. Un simulador también puede disponer de mecanismos para colocar el estado inicial más adecuado, evitando así las largas secuencias que pueden llegar a necesitarse hasta alcanzar dicho estado. Aparte de conseguir situar al circuito en un estado inicial, el tercer problema está en cómo observar el estado para verificar su exactitud. En algunos circuitos, es necesario aplicar una serie adicional de entradas para determinar el estado del circuito en un punto determinado. La alternativa más simple es preparar la simulación para que el estado del circuito pueda observarse directamente; la aproximación para hacer esto varía, dependiendo del simulador y de si el circuito tiene o no, jerarquía. Una solución algo ruda pero que funciona con todos los simuladores es añadir una salida al circuito con un camino desde cada señal de variable de estados.

Un último problema a debatir con más detalle es en qué instantes se han de aplicar las entradas y observar las salidas del circuito en relación al flanco activo del reloj. Inicialmente, discutimos estos tiempos para la *simulación funcional* que tiene como objeto determinar o comprobar la función del circuito. En la simulación funcional, los componentes del circuito no tienen ningún retardo o un retardo muy pequeño. Mucho más compleja resulta la *simulación temporal* en la que los elementos del circuito tienen retardos reales y cuyo objetivo es comprobar el funcionamiento apropiado del circuito en lo que se refiere a los tiempos.

Por defecto, en la simulación funcional, algunos simuladores emplean retardos muy pequeños en los componentes de modo que puedan apreciarse las variaciones de las señales con tal de que la escala de tiempos empleada en el gráfico sea suficientemente pequeña. Suponga una simulación para su circuito en la que tanto el retardo del componente como los tiempos de setup y hold para los flip-flops son todos de 0.1 ns y en la que el mayor retardo entre dos flancos positivos de reloj consecutivos es de 1.2 ns. Si para su simulación emplea un periodo de reloj de 1.0 ns, los resultados de la simulación serán erróneos siempre que el resultado dependa del retardo mayor. Así, para la simulación funcional con este simulador, debe escogerse un periodo de reloj mayor o debe cambiarse el retraso predefinido por el usuario por un valor menor.

Además del periodo del reloj, también es importante el instante en el que se aplican las entradas en relación al flanco positivo del reloj. Para la simulación funcional, con un retardo mínimo de componente predefinido, las entradas se deben presentar durante un ciclo de reloj antes del flanco positivo, siempre lo antes posible y mientras que el reloj todavía esté a 1. También es



□ **FIGURA 6-22**
Simulación temporal

el momento apropiado para cambiar los valores de las señales reset de modo que nos aseguremos de que la señal reset controla el estado en lugar de hacerlo el flanco del reloj o una combinación sin sentido de reloj y reset.

El último problema es el instante en el que se examina el resultado de la simulación funcional. Como muy tarde, las variables de estado y las salidas deben alcanzar sus valores finales antes del flanco positivo de reloj. Aunque es posible observar los valores en otros instantes, este momento proporciona un tiempo de observación seguro para la simulación funcional.

En la Figura 6-22 simplemente se resumen las ideas presentadas hasta ahora. Los cambios en las entradas en Reset e Input, encerradas un círculo azul, se producen alrededor del 25% del ciclo del reloj. Los valores de la señales State y Out, también encerradas en un círculo azul, se observan antes del 100% del ciclo del reloj.

6-5 DISEÑO DE CIRCUITOS SECUENCIALES

El diseño de circuitos secuenciales síncronos comienza a partir de una serie de especificaciones y culmina en un diagrama lógico o una lista de funciones booleanas de las que puede obtenerse el diagrama lógico. Frente a un circuito combinacional que se especifica totalmente por una tabla de verdad, un circuito secuencial necesita de una tabla de estados para su especificación. Así, el primer paso en el diseño de un circuito secuencial es obtener una tabla de estados o una representación equivalente como un diagrama de estados.

Un circuito secuencial síncrono se realiza a partir de flip-flops y de puertas combinacionales. El diseño del circuito consiste en elegir los flip-flops y encontrar una estructura combinacional que, junto con los flip-flops, produzca un circuito que cumpla las especificaciones dadas. El número mínimo de flip-flops viene determinado por el número de estados del circuito; n flip-flops pueden representar como mucho 2^n estados binarios. El circuito combinacional se deriva de la tabla de estados evaluando las ecuaciones de entrada y de salida de los flip-flops. De hecho, una vez determinado el tipo y el número de flip-flops, el proceso del diseño transforma un problema de circuitos secuenciales en un problema de circuitos combinacionales. De este modo, pueden aplicarse las técnicas de diseño de circuitos combinacionales.

Procedimiento del diseño

El siguiente procedimiento para el diseño de circuitos secuenciales es similar al empleado para los circuitos combinacionales pero con algunos pasos adicionales:

1. **Especificación:** escribir una especificación para el circuito, si aún no existe.
2. **Formulación:** obtener un diagrama de estados o una tabla de estados a partir de la especificación del problema.
3. **Asignación de estados:** si sólo ha sido posible obtener el diagrama de estados, obtener la tabla de estados. Asignar los códigos binarios a los estados de la tabla.
4. **Determinación de la ecuación de entrada al flip-flop:** seleccionar el tipo o tipos de flip-flops. A partir de la tabla de estados, obtener las ecuaciones de entrada de los flip-flops de las entradas codificadas del estado futuro.
5. **Determinación de la ecuación de salida:** obtener las ecuaciones de salida a partir de las salida descritas en la tabla de estados.
6. **Optimización:** optimizar las ecuaciones de entrada y de salida de los flip-flops.

7. **Mapeado tecnológico:** dibujar un diagrama lógico del circuito empleando flip-flops, ANDs, ORs, e inversores. Transformar el diagrama lógico en un nuevo diagrama que emplee los flip-flops y puertas disponibles en la tecnología.
8. **Comprobación**: verificar la corrección del diseño final.

Por conveniencia, usualmente omitiremos el mapeado tecnológico del paso 7 y utilizaremos sólo flip-flops, puertas AND, puertas OR, e inversores en el esquemático.

Localización de los diagramas de estados y las tablas de estados

La especificación de un circuito a menudo es una descripción verbal del comportamiento del circuito. Hay que interpretar esta descripción para encontrar un diagrama de estados o la tabla de estados del paso de formulación en el procedimiento del diseño. A menudo, esta es la parte más creativa del diseño, ya que muchos de los otros pasos se realizarán automáticamente mediante herramientas de diseño asistido por computadora.

El principio para formular tablas y diagramas de estados es la comprensión intuitiva del concepto de estado. Un estado se emplea para «recordar» qué combinaciones de entradas se han aplicado al circuito en cualquier flanco activo o durante cualquier pulso activo de reloj. En algunos casos, los estados pueden guardar literalmente los valores de la entrada manteniendo una historia completa de la secuencia que ha ido apareciendo en sus entradas. En la mayoría de los casos, sin embargo, un estado es una *abstracción* de la sucesión de combinaciones de la entrada en los instantes de disparo. Por ejemplo, un determinado estado S_1 puede representar el hecho que entre la secuencia de valores aplicados sobre una entrada X de un único bit «el valor 1 aparezca en X durante los tres últimos flancos consecutivos de reloj». Así, el circuito estaría en estado S_1 después de la serie... 00111 o ... 0101111, pero no estaría en S_1 después de la serie... 00011 o ... 011100. Un estado S_2 podría representar el hecho que la secuencia de combinaciones de entrada de 2 bits aplicadas «está en orden 00, 01, 11, 10 permitiendo cualquier número de repeticiones consecutivas de cada combinación y siendo 10 la combinación más recientemente aplicada». El circuito estaría en S_2 para las siguientes secuencias del ejemplo: 00, 00, 01, 01, 01, 11, 10, 10 o 00, 01, 11, 11, 11, 10. El circuito no estaría en el estado S_2 para las secuencia: 00, 11, 10, 10 o 00, 00, 01, 01, 11, 11. Para formular el diagrama de estados o la tabla de estados es útil anotar la abstracción que representa cada estado. En algunos casos, es más fácil describir la abstracción refiriéndose a los valores que se han producido tanto en las salidas como en las entradas. Por ejemplo, el estado S_3 podría representar la abstracción que «el bit de salida Z_2 es 1 y la combinación de entrada tiene el bit X_2 a 0». En este caso, Z_2 igual a 1 podría representar una serie compleja de secuencias pasadas de combinaciones de entrada que serían mucho más difíciles de describir en detalle.

Cuando se formula una tabla de estados o un diagrama de estados, se añaden nuevos estados. ¡Es posible hacer innecesariamente grande o incluso infinito el tamaño de la serie de estados! En lugar de añadir un nuevo estado para cada estado actual y su posible combinación de entrada, es esencial que los estados se reutilicen como próximos estados para prevenir el crecimiento desenfrenado de estados tal y como hemos descrito anteriormente. El mecanismo para realizar esto es conocer la abstracción que cada estado representa. Para ilustrarlo, considere el estado S_1 definido previamente como una abstracción «el valor 1 ha aparecido a los últimos tres flancos de reloj». Si se ha entrado en S_1 debido a la sucesión ... 00111 y la próxima entrada es un 1, dando la sucesión ... 001111 ¿se necesita un nuevo estado o el próximo estado puede ser S_1? Examinando la nueva sucesión, vemos que los últimos tres valores de la entrada son 1 tal y como define la abstracción para el estado S_1. Así que, el estado S_1 se puede emplear como el

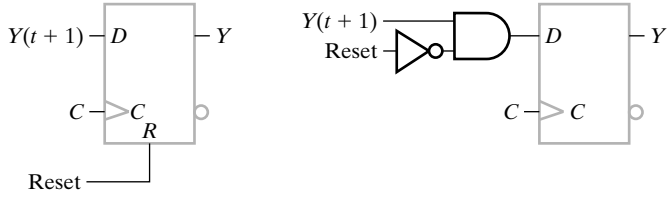
próximo estado para el estado actual S_1 y para el valor de entrada 1, evitando la definición de un nuevo estado.

Cuando se desconoce el estado de los flip-flops en el instante del encendido del circuito, es costumbre, por lo menos en algunos de los circuitos proporcionar una señal general de reset para inicializar el estado de los flip-flops. Esto evita que dichos circuitos arranquen en un estado desconocido. Típicamente, la señal general de reset se aplica a las entradas asíncronas de los flip-flops (*véase* la Figura 6-23) antes de que comience el funcionamiento síncrono. En muchos casos, los flip-flops son reseteados a 0, pero algunos pueden ponerse a 1, dependiendo de cuál sea su estado inicial deseado. Si hay un elevado número de flip-flops, y dependiendo del funcionamiento concreto del circuito, podrán inicializarse sólo algunos de ellos.

Cuando se enciende un sistema digital por primera vez se desconoce el estado de los flip-flops. Puede aplicarse una secuencia de entradas con el circuito en un estado desconocido con tal que dicha sucesión coloque al circuito en un estado conocido antes que se esperen salidas significativas. De hecho, muchos de los circuitos secuenciales que diseñaremos en los siguientes capítulos serán de este tipo. En este capítulo, sin embargo, los circuitos diseñados partirán de un *estado inicial conocido*, y además, contarán con un mecanismo hardware que permita colocar al circuito en dicho estado desde un estado desconocido. Este mecanismo es un *reset* o *señal general de reset*. Sin tener en cuenta las restantes entradas aplicadas al circuito, el reset coloca al circuito en su estado inicial. De hecho, el estado inicial se llama a menudo *estado reset*. Normalmente, la señal reset se activa automáticamente cuando el circuito se enciende. Además, puede activarse electrónicamente o pulsando un botón de inicialización.

El reset puede ser asíncrono, teniendo lugar sin necesidad de activar el reloj. En este caso, el reset se aplica a las entradas asíncronas de los flip-flops del circuito. Tal y como se muestra en la Figura 6-23(a). Este diseño asigna 00...0 al estado inicial de los flip-flops para resetearlos. Si se desea un estado inicial con un código diferente, entonces la señal Reset puede conectarse selectivamente a las entradas de set asíncronas en lugar de a las entradas de reset asíncronas. Es importante tener en cuenta que estas entradas no deben usarse en el diseño del circuito síncrono normal. En cambio, están reservadas para un reset asíncrono que devuelve el sistema a un estado inicial. Empleando estas entradas asíncronas como una parte del diseño del circuito síncrono se viola la definición fundamental de circuito síncrono, ya que se permite a un flip-flop cambiar asíncronamente de estado.

Alternativamente, el reset puede ser síncrono necesitando de un disparo de reloj para que se produzca. Debemos incorporar el reset en el diseño síncrono del circuito. Una sencilla aproximación del reset síncrono para flip-flops *D*, que no incluye el bit de reset en la combinación de entrada, es añadir la puerta AND mostrada en la Figura 6-23(b) después de realizar el diseño normal del circuito. Este diseño también asigna 00...0 al estado inicial. Si se desea un código de estados inicial diferente, entonces la puerta AND y el inversor para Reset se sustituye por una puerta OR con la entrada Reset.



(a) Reset asíncrono (b) Reset síncrono

□ **FIGURA 6-23**
Reset asíncrono y síncrono para un flip-flop *D*

Los siguientes dos ejemplos ilustran el proceso de formulación, produciendo cada uno de ellos un estilo diferente de diagrama de estados.

EJEMPLO 6-2 Encontrar un diagrama de estados para un detector de secuencia

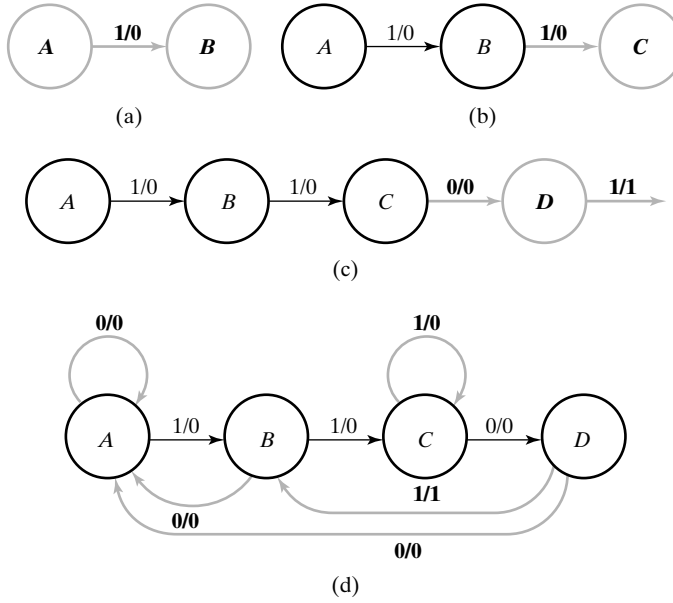
El primer ejemplo es un circuito que reconoce una determinada secuencia de bits, dentro de una secuencia más larga. Se trata de «un detector de secuencia» que tiene una entrada *X* y una salida *Z*. Cuenta con un Reset aplicado a las entradas asíncronas reset de sus flip-flops para inicializar el estado del circuito a ceros. El circuito detecta la secuencia de bits 1101 en *X* haciendo *Z* igual a 1 cuando las tres entradas anteriores al circuito sean 110 y la señal de entrada sea 1. En cualquier otro caso, *Z* será igual a 0.

El primer paso en el proceso de la formulación es determinar si el diagrama o la tabla de estados deben ser Autómatas de Mealy o de Moore. La parte de la especificación anterior que dice «... haciendo *Z* igual a 1 cuando las tres entradas anteriores al circuito son 110 y la señal de entrada es un 1» implica que la salida es función no sólo del estado actual, sino también de la señal de alimentación. En consecuencia, se necesita un modelo de Autómata Mealy en el que la salida depende tanto del estado como de las entradas.

Tenga presente que el factor principal en la formulación de cualquier diagrama de estados es reconocer que se emplean los estados para «recordar» la historia de las entradas. Por ejemplo, en la secuencia 1101, para poder generar el valor de salida 1 al mismo tiempo que el 1 del final de la secuencia, el circuito debe estar en un estado que «recuerde» que las tres entradas anteriores eran 110. Con este concepto en mente, empezamos a formular el diagrama de estados definiendo un estado inicial arbitrario *A* como el estado de reset, el estado en el que «aún no ha ocurrido ninguna secuencia». Si aparece un 1 en la entrada, este evento ha de «recordarse» ya que el 1 es el primer bit de la secuencia, y el estado después del pulso del reloj no puede ser *A*. Entonces, se establece un segundo estado, el *B*, para representar la ocurrencia del primer 1 en la sucesión. Además, se coloca una transición desde *A* hasta *B* que se etiqueta con un 1 para representar la ocurrencia del primer 1 en la secuencia. Puesto que éste no es el último 1 de la secuencia, su salida será un 0. Esta parte del diagrama de estados se muestra en la Figura 6-24(a).

El siguiente bit en la secuencia es un 1. Cuando se produce este 1 en el estado *B*, se necesita un nuevo estado para representar la ocurrencia de dos 1 seguidos en la secuencia de entrada, es decir, la ocurrencia de un 1 adicional estando en el estado *B*. Entonces, se añade un estado *C* y la transición asociada, tal y cómo se representa en la Figura 6-24(b). El próximo bit de la secuencia es un 0. Cuando aparece este 0 en el estado *C*, se necesita un nuevo estado para representar la ocurrencia de los dos 1 seguida por un 0. Por tanto, se añade el estado adicional *D* con una transición que tiene una entrada 0 y una salida 0. Dado que el estado *D* representa la ocurrencia 110 como los valores de los tres bits de entrada en *X*, la ocurrencia de un 1 en el estado *D* completaría la secuencia a reconocer, de modo que la transición para el valor de entrada 1 del estado *D* tendría un valor de salida de 1. En la Figura 6-24(c) se muestra el diagrama de estados parcial resultante que representa completamente la ocurrencia de la secuencia a detectar.

Observe en la Figura 6-24(c) que, para cada estado, se especifica una única transición para uno sólo de los dos posibles valores de entrada. Tampoco se define todavía el estado destino de la transición *D* para la entrada 1. Las restantes transiciones se basan en la idea de que el detector de secuencia ha de identificar la sucesión 1101, sin tener en cuenta cuándo se produce dentro de una secuencia más larga. Suponga que se representa una parte inicial de la secuencia 1101 por un estado en el diagrama. Entonces, la transición de este estado para un valor de entrada que represente el próximo valor en la secuencia debe dirigirse hacia un estado tal que su



□ **FIGURA 6-24**
Construcción de un diagrama de estados para el Ejemplo 6.2

salida sea 1 si se aplican los restantes bits de la secuencia a detectar. Por ejemplo, el estado *C* representa los primeros dos bits, 11, de la secuencia 1101. Si los próximos valores de entrada son 0, entonces se entra en un estado, en este caso, el *D*, que presenta un 1 en la salida si se aplica el bit restante de la secuencia, el 1.

Luego, evalúa donde va la transición desde el estado *D* para un 1 en la entrada. Puesto que la entrada de la transición es un 1, podría ser el primero o el segundo de los bits en la secuencia a detectar. Pero, dado que el circuito está en el estado *D*, es evidente que la anterior entrada fue un 0. Entonces, este 1 de entrada es el primero 1 de la secuencia, ya que no puede estar precedido por otro 1. El estado que representa la ocurrencia del primer 1 en la secuencia es el *B*, entonces la transición del estado *D* para la entrada 1 es *B*. Esta transición se representa en el diagrama de la Figura 6-24(d). Examinando el estado *C*, podemos volver atrás a través de los estados *B* y *A* para ver cómo la ocurrencia de una entrada 1 en *C* es, por lo menos, el segundo 1 en la secuencia. El estado que representa la ocurrencia de dos 1 en la secuencia es el *C*, de modo que la nueva transición es al estado *C*. Puesto que la combinación de dos 1 no es la secuencia a reconocer, la salida para la transición es 0. Repitiendo este mismo análisis para las transiciones perdidas desde los estados *B* y *A*, se obtiene el diagrama de estados final de la Figura 6-24(d). La tabla resultante se da en forma bidimensional en la Tabla 6.3. ■

Un problema que surge en la formulación de cualquier diagrama de estados es si, a pesar de los mayores esfuerzos de diseñador, se utilizan un número excesivo de estados. Éste no es el caso del ejemplo anterior, puesto que cada estado representa la historia de la entrada, que es esencial para el reconocimiento de la secuencia formulada. No obstante, si hay excesivos estados, sería deseable combinar los estados en los menos posibles. Esto se puede hacer utilizando métodos *ad hoc* y procedimientos formales de minimización de estados. Debido a la complejidad de esto último, particularmente en el caso en el que aparecen condiciones de indiferencia en la tabla de estados, no se tratan aquí estos procedimientos formales. Para los estudiantes que

□ **TABLA 6-3**
Tabla de estados para el diagrama de estado de la Figura 6-21

Estado actual	Estado futuro		Salida	
	$X = 0$	$X = 1$	$X = 0$	$X = 1$
A	*A*	*B*	0	0
B	*A*	*C*	0	0
C	*D*	*C*	0	0
D	*A*	*B*	0	1

estén interesados, los procedimientos de minimización de estados se encuentran en las referencias enumeradas al final del capítulo. En el siguiente ejemplo se ilustra un método adicional para evitar estados extras.

EJEMPLO 6-3 Encontrar el diagrama del estado para decodificador BCD a exceso-3

En el Capítulo 3, se diseñó un decodificador BCD a exceso-3. En este ejemplo, la función del circuito es similar sólo que las entradas, en lugar de presentarse simultáneamente al circuito, se presentan consecutivamente en ciclos de reloj sucesivos, empezando por el bit menos significativo. En la Tabla 6-4(a), se listan las secuencias de entrada y las secuencias de salida correspondientes comenzando por el bit de menor peso. Por ejemplo, durante cuatro ciclos del reloj consecutivos, si se aplican 1010 a la entrada, la salida será 0001. Con el fin de generar cada bit de salida en el mismo ciclo de reloj que el bit de entrada correspondiente, la salida dependerá tanto del valor de entrada actual como del estado. Las especificaciones también establecen que

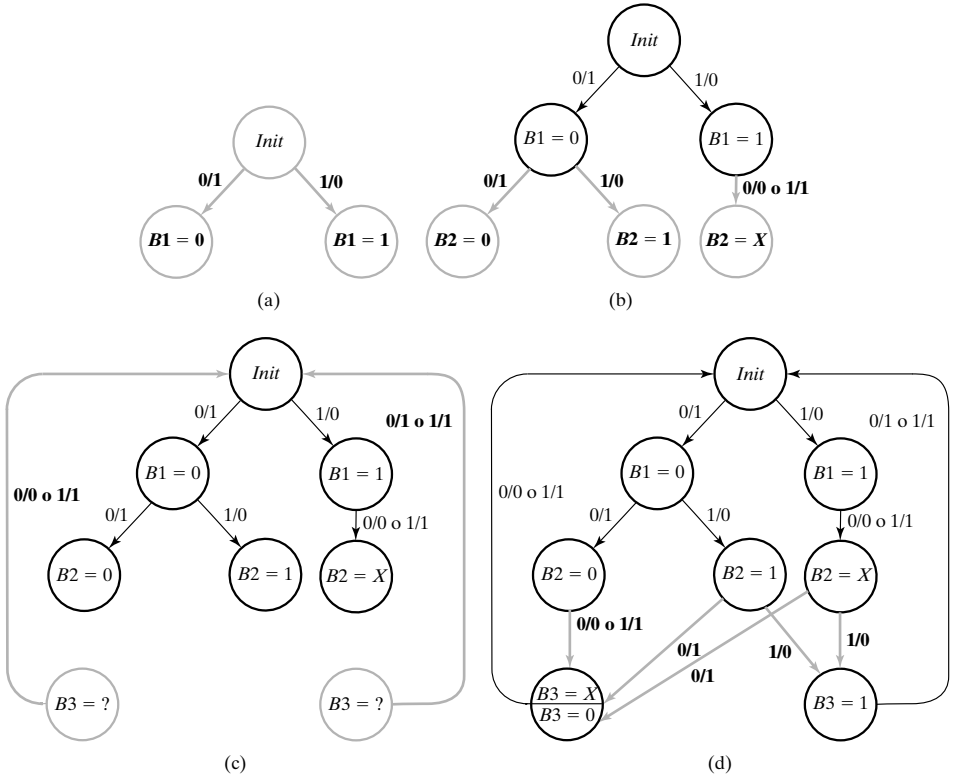
□ **TABLA 6-4**
Tabla de secuencias para el ejemplo del convertidor de código

(a) Secuencias ordenadas por dígito representado								(b) Secuencias ordenadas por orden de bits iniciales							
Entrada BCD				Salida exceso-3				Entrada BCD				Salida exceso-3			
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0
1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	1
0	1	0	0	1	0	1	0	0	0	1	0	1	1	1	0
1	1	0	0	0	1	1	0	0	1	0	0	1	0	1	0
0	0	1	0	1	1	1	0	0	1	1	0	1	0	0	1
1	0	1	0	0	0	0	1	1	0	0	0	0	0	1	0
0	1	1	0	1	0	0	1	1	0	0	1	0	0	1	1
1	1	1	0	0	1	0	1	1	0	1	0	0	0	0	1
0	0	0	1	1	1	0	1	1	1	0	0	0	1	1	0
1	0	0	1	0	0	1	1	1	1	1	0	0	1	0	1

el circuito debe estar listo para recibir una nueva secuencia de 4 bits en cuanto se haya completado la secuencia anterior. La entrada a este circuito se etiqueta con *X* y la salida se etiqueta con *Z*. Con el fin de centrarnos en las combinaciones pasadas de las entradas, las filas de la Tabla 6-4(a) se ordenan según el valor del primer bit, el valor del segundo bit, y el valor del tercer bit de la secuencia de entrada. Así resulta la Tabla 6-4(b).

El diagrama de estados comienza con un estado inicial representado en la Figura 6-25(a). Al examinar la primera columna de bits de la Tabla 6-4(b) se aprecia que un 0 produce un salida 1 y un 1 produce una salida 0. Ahora nos preguntamos ¿necesitamos recordar el valor del primer bit? En la Tabla 6-4(b), cuando el primer bit es un 0, un 0 en el segundo bit genera un 1 en la salida y mientras que un 1 en el segundo bit genera un 0 en la salida. Por el contrario, si el primer bit es un 1, un 0 en el segundo bit origina una salida 0 mientras que un 1 en el segundo bit genera una salida de 1. Está claro que la salida para el segundo bit no puede determinarse sin «recordar» el valor del primer bit. Así, la primera entrada igual a 0 y la primera entrada igual a 1 deben ser dos estados diferentes tal y como se refleja en la Figura 6-25(a), donde también se muestran los valores de entrada/salida sobre las líneas de dirección hacia los nuevos estados.

Luego, hay que determinar si las entradas siguientes a los dos nuevos estados necesitan dos estados para «recordar» el valor del segundo bit. En las dos primeras columnas de entrada de la Tabla 6-4(b), la secuencia 00 produce las salidas para el tercer bit siendo 0 para la entrada 0 y 1 para entrada 1. Por otro lado, para la secuencia de entrada 01, la salida para el tercer bit es 1 para la entrada 0 y 0 para la entrada 1. Puesto que son diferentes para los mismos valores de entrada en el tercer bit, se necesitan estados distintos, tal y como se ilustra en la Figura 6-25(b).



□ **FIGURA 6-25**
Construcción del diagrama de estados para el Ejemplo 6.3

Un análisis similar para las secuencias de entrada 10 y 11 en el que se han examinado las salidas para el tercer y cuarto bit, muestra que el valor del segundo bit no tiene efecto en los valores de salida de dichos bits. Por todo ello, en la Figura 6-25(b) hay un único estado futuro para el estado $B1 = 1$.

A estas alturas, pueden resultar seis potenciales nuevos estados a partir de los tres estados añadidos. Observe, sin embargo, que estos estados sólo necesitan definir las salidas para el cuarto bit puesto que el siguiente estado después de esto será *Init* preparándose para recibir la próxima secuencia de entrada de 4 bits ¿Cuántos estados se necesitarán para especificar las diferentes posibilidades para el valor de salida del último bit? Mirando la última columna, una entrada de 1 siempre produce un 1 de salida mientras que un 0 puede producir un 0 ó un 1 de salida. De esta forma, como mucho se requieren dos estados, uno que genera una salida 0 para una entrada 0 y otro que genera una salida 1 para una entrada 0. La salida para la entrada 1 es la misma para los dos estados. En la Figura 6-25(c) se han añadido estos dos estados. Para que el circuito esté preparado para recibir la siguiente secuencia, el estado siguiente a estos dos estados en *Init*.
Finalmente falta determinar los arcos azules de la Figura 6-25(d). Los arcos desde cada estado *B*2 se han definido en base al tercer bit en la secuencia de entrada/salida. El próximo estado se puede elegir en función de la respuesta a la entrada 0 en el bit cuarto de la secuencia. El estado *B*2 alcanza al estado *B*3 de la izquierda con $B3 = 0$ o $B3 = 1$ indicado por $B3 = X$ en la mitad superior del estado de *B*3. Los otros dos estados de B2 alcanzan este mismo estado con $B3 = 1$ como se indica en la mitad inferior del estado. Estos mismos dos estados *B*2 alcanzan al estado *B*3 a la derecha con $B3 = 0$ tal y como representa la etiqueta en el estado. ■

Asignación de estados

Frente a los estados de los ejemplos analizados, en los diagramas que se han construido se han asignado nombres simbólicos para los estados en lugar de códigos binarios. Es necesario reemplazar estos nombres simbólicos con códigos binarios con el fin de proceder con el diseño. En el general, si hay m estados, entonces los códigos deben contener n bits dónde $2^n \geqslant m$, y cada estado debe asignarse a un único código. Así que, para el circuito de la Tabla 6-3 con cuatro estados, los códigos asignados a los estados necesitarán de dos bits.

Comenzamos asignando un código al estado inicial reset. Si las primeras cuatro entradas al circuito son 1101 después de *Reset* = 1, deberían detectarse. Pero si ocurre que la primera secuencia de entrada es 101, 01, o 1, no debería ser detectada. El único estado que puede proporcionar esta propiedad es el estado *A*. Entonces, debe asignarse el código 00 al estado *A* con ayuda de las entradas asíncronas de reset de los flip-flops. Como base para asignar un código a los estados restantes, existe un extenso trabajo en la asignación de códigos a los estados, pero es demasiado complejo para tratarlo aquí. Estos métodos se han centrado principalmente en intentar seleccionar los códigos de tal manera que se minimice la lógica necesaria para implementar las ecuaciones de entrada y salida de los flip-flops. En nuestro ejemplo, simplemente asignamos los códigos de estados siguiendo el orden del Código Gray, empezando con el estado *A*. El Código Gray se selecciona en este caso para simplificar más fácilmente el estado futuro y la función de salida en el Mapa de Karnaugh. La tabla de estados con los códigos asignados se muestra en la Tabla 6-5.

Diseñando con flip-flops *D*

El resto del procedimiento para diseñar el circuito secuencial se muestra con el próximo ejemplo. Queremos diseñar un circuito secuencial síncrono que funciona según la Tabla de esta-

□ **TABLA 6-5**
Tabla 6-3 con los nombres reemplazados por códigos binarios

Estado actual	Estado futuro		Salida	
AB	***X* = 0**	***X* = 1**	***X* = 0**	***X* = 1**
00	00	01	0	0
01	00	11	0	0
11	10	11	0	0
10	00	01	0	1

dos 6-2, el detector de secuencia de la Tabla 6-5. Esta tabla de estados, con los códigos binarios asignados a los estados, especifica cuatro estados, dos valores de entrada, y dos valores de salida. Se necesitan dos flip-flops para representar los cuatro estados. Etiquetamos las salidas de los flip-flops con las letras *A* y *B*, la entrada con *X*, y la salida con *Z*.

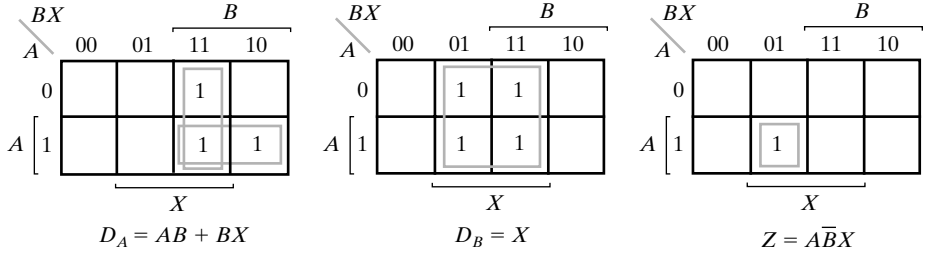
Para este circuito, ya se han completado los pasos 1, 2 y 3 del procedimiento de diseño. El paso 4 comienza escogiendo flip-flops tipo *D*. Para completar el paso 4, se obtienen las ecuaciones de entrada de los flip-flops a partir de los valores del estado futuro que se listan en la tabla. En el paso 5 se obtiene la ecuación de salida a partir de los valores de *Z* que se listan en la misma tabla. Las ecuaciones de la entrada y de la salida de los flip-flops pueden obtenerse como suma de minitérminos de las variables del estado actual *A* y *B* y de la variable de entrada *X*:

$$A(t+1) = D_A(A, B, X) = \Sigma m(3, 6, 7)$$
$$B(t+1) = D_B(A, B, C) = \Sigma m(1, 3, 5, 7)$$
$$Z(A, B, X) = \Sigma m(5)$$

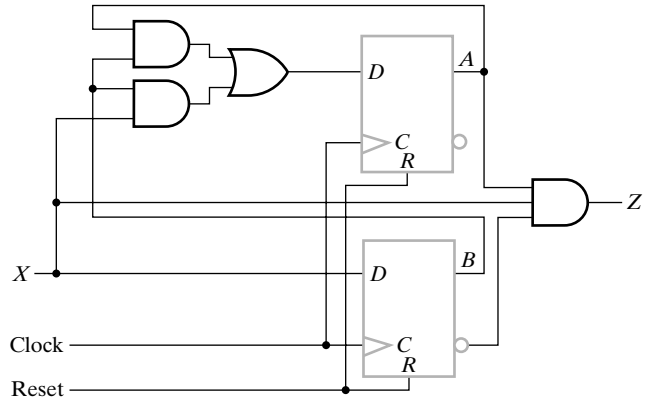
Las funciones booleanas se simplifican usando los Mapas de Karnaugh trazados en la Figura 6-26. Las funciones simplificadas son

$$D_A = AB + BX$$
$$D_B = X$$
$$Z = A\bar{B}X$$

El diagrama lógico del circuito secuencial se muestra en la Figura 6-27.



□ **FIGURA 6-26**
Mapas de Karnaugh para las ecuaciones de entrada de los flip-flops y para la salida *Z*



□ **FIGURA 6-27**
Diagrama lógico para el circuito secuencial con flip-flops tipo *D*

Diseñando con estados no usados

Un circuito con *n* flip-flops tiene 2^n estados binarios. Sin embargo, la tabla de estados de la que se deriva originalmente el circuito puede tener cualquier número *m* de estados con tal que $m \leqslant 2^n$. Los estados que no se utilizan en un circuito secuencial concreto no aparecen en la tabla de estados. Al simplificar las ecuaciones de entrada, los estados sin usar pueden tratarse como condiciones indiferentes. La tabla de estados de la Tabla 6-6 define tres flip-flops, *A*, *B* y *C*, y una entrada, *X*. No hay ninguna columna de salida lo que significa que los flip-flops sirven como salidas del circuito. Con tres flip-flops es posible especificar hasta ocho estados, pero los estados listados en la tabla de estados son sólo cinco. Hay tres estados sin usar que no se incluyen en la tabla, son: 000, 110 y 111. Cuando se incluye una entrada de 0 o 1 como valor de estado actual, se obtienen seis combinaciones sin usar para el estado actual y columnas de la entrada: 0000, 0001, 1100, 1101, 1110 y 1111. Estas seis combinaciones no se listan en la tabla de estados e incluso pueden tratarse como mini términos indiferentes.

□ **TABLA 6-6**
Tabla de estados para un diseño con estados no utilizados

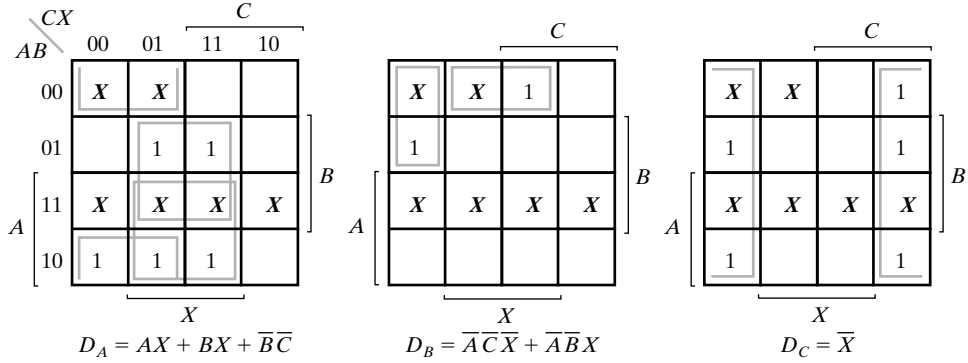
Estado actual			**Entrada**	**Estado futuro**		
A	***B***	***C***	***X***	***A***	***B***	***C***
0	0	1	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	0	1	1
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	1	0	0
1	0	0	0	1	0	1
1	0	0	1	1	0	0
1	0	1	0	0	0	1
1	0	1	1	1	0	0

Las tres ecuaciones de entrada para los flip-flops *D* se obtienen a partir de los valores del estado futuro y se simplifican en los Mapas de Karnaugh de la Figura 6-28. Cada mapa tiene seis minitérminos indiferentes en los cuadrados que corresponden a los binarios de 0, 1, 12, 13, 14 y 15. Las ecuaciones optimizadas son:

$$D_A = AX + BX + \bar{B}\bar{C}$$
$$D_B = \bar{A}\bar{C}\bar{X} + \bar{A}\bar{B}X$$
$$D_C = \bar{X}$$

El diagrama lógico puede obtenerse directamente de las ecuaciones de entrada y no se dibujará aquí.

Es posible que una interferencia externa o un funcionamiento defectuoso lleven al circuito a uno de los estados sin usar. Por tanto, a veces es deseable especificar, totalmente o por lo menos parcialmente, los próximos valores de estados o de salida para los estados sin usar. Dependiendo de la función y de la aplicación del circuito, pueden aplicarse varias ideas. Primero, pueden especificarse las salidas para los estados sin usar para que cualquier acción que resulte de la entrada y de las transiciones entre estados sin usar no sea perjudicial. Segundo, se puede proporcionar una salida adicional o un código de la salida sin usar para indicar que el circuito ha entrado en un estado incorrecto. Tercero, para asegurar que es posible volver al funcionamiento normal sin tener que resetear el sistema entero, se puede especificar el comportamiento del estado futuro para aquellos estados sin usar. Normalmente, los próximos estados se seleccionan de modo que se alcanza uno de los estados en unos pocos ciclos del reloj, sin tener en cuenta los valores de entrada. La decisión acerca de cuál de las tres opciones se va a aplicar, individualmente o en combinación, es función de la aplicación del circuito o de las políticas de cada grupo de diseño.



□ **FIGURA 6-28**
Mapas para optimización de ecuaciones

Verificación

Los circuitos secuenciales se pueden verificar demostrando que el circuito produce el diagrama de estados original o la tabla de estados. En los casos más sencillos se aplican todas las posibles combinaciones de entrada con el circuito en cada uno de los estados y se observan las variables de estados y las salidas. En circuitos pequeños, la comprobación real se puede realizar a mano. Lo más habitual es recurrir a la simulación. En la simulación manual, es válido aplicar cada una

de las combinaciones estado-entrada y verificar que la salida y el próximo estado son los correctos.

La verificación mediante la simulación es menos tediosa, pero usualmente requiere una sucesión de combinaciones de entrada y aplicar una señal de reloj. Con el fin de verificar una combinación estado-entrada, primero hay que aplicar una sucesión de combinaciones de entrada para poner al circuito en el estado deseado. Es muy útil encontrar una única secuencia para probar todas las combinaciones de estados entrada. El diagrama de estados es ideal para generar y perfeccionar dicha sucesión. Se debe generar una secuencia para aplicar cada combinación de entrada en cada estado mientras se observa la salida y el estado que aparece tras el flanco del reloj positivo. La longitud de la secuencia puede perfeccionarse empleando el diagrama de estados. La señal de reset se empleará como entrada durante esta sucesión. En particular, se utiliza para resetear el circuito llevándolo a su estado inicial.

En el Ejemplo 6-4, se ilustran tanto la verificación manual como la verificación basada en simulación.

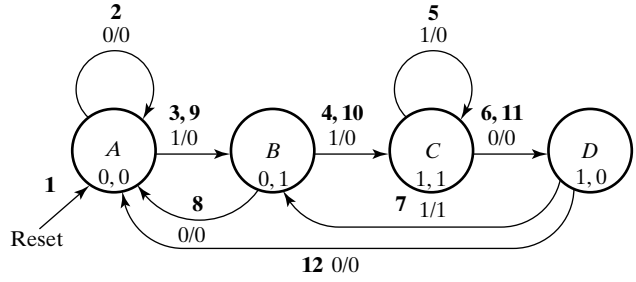
EJEMPLO 6-4 Verificación del detector de secuencia

En la Figura 6-24(d) aparece el diagrama de estados para el detector de secuencia y en la Figura 6-27 se muestra el diagrama lógico. Hay cuatro estados y dos combinaciones de entrada, dando un total de ocho combinaciones estado-entrada para verificar. El estado siguiente se presenta en las salidas de los flip-flops después del flanco positivo de reloj. Para los flip-flops *D*, el estado siguiente será simplemente igual que la entrada *D* antes del flanco de reloj. Para otros tipos de flip-flops, las entradas a los flip-flops se emplean para determinar el próximo estado antes del flanco de reloj. Inicialmente, partiendo con el circuito en un estado desconocido, aplicamos un 1 a la entrada Reset. Esta entrada va a la entrada asíncrona de reset de los dos flip-flops de la Figura 6-27. Puesto que no hay ningún círculo en estas entradas, el valor 1 resetea ambos flip-flops a 0, dando el estado *A*(0, 0). A continuación, aplicamos la entrada 0, y mediante simulación manual en el circuito de la Figura 6-27, encontramos que la salida es 0 y el próximo estado es *A*(0, 0) que está de acuerdo con la transición para la entrada 0 desde el estado *A*. Continuando con la simulación, la entrada 1 desde el estado *A* lleva al próximo estado *B*(0, 1) con salida 0. Para el estado *B*, una entrada 0 da salida 0 y un próximo estado *A*(0, 0), y una entrada 1 da salida 0 y un siguiente estado *C*(1, 1). Este mismo proceso puede continuarse para cada una de las dos combinaciones de la entrada para los estados *C* y *D*.

Para verificar mediante simulación, se genera una secuencia de entradas que aplica todas las combinaciones estado-entrada y se acompaña por las secuencias de salida y de estados para verificar la salida y valor del estado futuro. La optimización requiere que el número de periodos de reloj empleados exceda del número de combinaciones estado-entrada (es decir, la repetición de combinaciones de pares estado-entrada se debe minimizar). Esto puede interpretarse como dibujar el camino más corto a través del diagrama de estados que atraviesa por lo menos una vez cada combinación estado-entrada.

Por conveniencia, en la Figura 6-29(a), se muestran los códigos para los estados y el camino a través del diagrama se denota por una sucesión de números enteros en azul que empiezan en el 1. Estos enteros corresponden al número de flancos positivos de reloj de la Figura 6-29(b), dónde se desarrolla la sucesión de la comprobación.

Los valores mostrados para el número de flancos de reloj representan los flancos ocurridos hasta justo antes del flanco positivo del reloj (es decir, durante el intervalo de tiempo de setup). El flanco de reloj 0 ocurre en $t = 0$ en la simulación y produce estados indeterminados en todas las señales. Se comienza con un valor 1 aplicado a la entrada Reset (1) para colocar al circuito



(a)

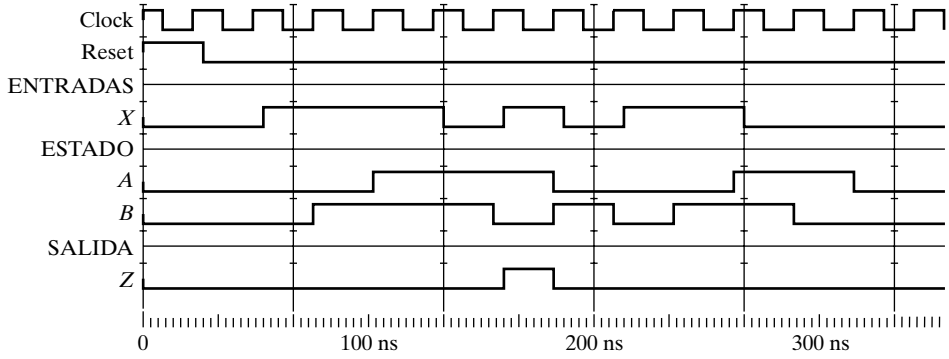
Clock Edge:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Input *R*:	X	1	0	0	0	0	0	0	0	0	0	0	0	
Input *X*:	X	0	0	1	1	1	0	1	0	1	1	0	0	
State (*A*, *B*):	X,X	0,0*	0,0	0,0	0,1	1,1	1,1	1,0	0,1	0,0	0,1	1,1	1,0	0,0
Output *Z*:	X	0	0	0	0	0	0	1	0	0	0	0	0	

(b)

□ **FIGURA 6-29**
Generación de una secuencia de test para la simulación del Ejemplo 6.5

en el estado *A*. Inicialmente se aplica la entrada 0 (2) de modo que el circuito permanece en *A*, después se aplica un 1 (3) comprobando la segunda combinación para el estado *A*. Ahora, en el estado *B* podemos tanto avanzar a *C* como volver a *A*. No está claro cual el la mejor decisión, de modo que de forma arbitraria se aplica un 1 (4) y se va al estado *C*. En el estado *C* se aplica un 1 (5) de modo que se permanece en *C*. Después se aplica un 0 para verificar la última entrada en el estado *C*. Ahora en *D*, debe decidirse si volver a *A* o a *B*. Si se retorna a *B* aplicando un 1 (7) entonces podemos chequear la transición de *B* a *A* para la entrada 0 (8), Entonces la transición que falta en el estado *D* es para la entrada 0. Para llegar al estado *D* desde *A* debe aplicarse la secuencia 1, 1, 0, (9) (10) (11) y después aplicar un 0 (12) para verificar la transición de *D* a *A*. Se han verificado 8 transiciones empleando una secuencia formada por un reset más 11 entradas. Aunque esta secuencia es óptima, el procedimiento empleado no asegura su optimalidad. Aun así, este procedimiento usualmente obtiene secuencias eficientes.

Para simular el circuito, capturamos el esquemático de la Figura 6-27 empleando el editor de esquemas Xilinx ISE 4.2 e introducimos la secuencia de la Figura 6-29(b) como una forma de onda empleando el Xilinx ISE 4.2 HDL Bencher. Al generar la forma de onda es importante que la entrada *X* cambie bastante antes del flanco de reloj. Esto asegura que hay tiempo disponible para visualizar la salida actual y para permitir que los cambios de entrada se propaguen a las entradas de los flip-flops antes de que comience el tiempo de setup. Esto se ilustra en las formas de onda de ENTRADA en la Figura 6-30 en las que *X* cambia un poco después del flanco positivo de reloj proporcionando una buena fracción del periodo de reloj para que el cambio se propague a los flip-flops. El circuito se simula con el simulador ModelSim MTI. Podemos comparar los valores justo antes del flanco positivo de reloj en los cronogramas de ESTADO y SALIDA de la Figura 6-30 con los valores mostrados en el diagrama de estados para cada periodo de reloj de la Figura 6-29. En este caso, la comparación verifica que el funcionamiento del circuito es correcto.



□ **FIGURA 6-30**
Simulación para el Ejemplo 6.5 ■

6–6 OTROS TIPOS DE FLIP-FLOPS

En esta sección se introducen los flip-flops *JK* y *T* y las representaciones de su comportamiento que se emplean en el análisis y el diseño.

Debido a su menor importancia en el diseño actual frente a los flip-flops tipo *D*, en el sitio Web del texto se dan ejemplos del análisis y del diseño que ilustran su uso.

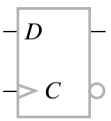
Flip-flops *JK* y *T*

En la Tabla 6-7 se muestran las características de cuatro tipos de flip-flops, el *SR* y el tipo *D* referenciados en la Sección 6-3, y el *JK* y *T* introducidos aquí. Con la excepción del flip-flop *SR* que es el maestro-esclavo, se muestra el símbolo para flip-flops activos por flanco positivo. También se muestra el diagrama lógico para implementar cada uno de los tipos de flip-flops dados. Un nuevo concepto, la *tabla característica*, define las propiedades lógicas de funcionamiento del flip-flop en forma tabular. Específicamente, la tabla define el próximo estado como una función del estado actual y de las entradas. $Q(t)$ se refiere al estado actual previo a la aplicación de un pulso del reloj. $Q(t + 1)$ representa el estado un periodo de reloj después (es decir, es el próximo estado). Observe que el flanco de disparo (o pulso) de la entrada *C* no se lista en la tabla característica, pero se supone que ocurre entre el tiempo t y el tiempo $t + 1$. Junto a la tabla característica se muestra la ecuación característica para cada tipo de flip-flop. Estas ecuaciones definen el próximo estado después del pulso de reloj para cada uno de los flip-flops como una función de las entradas actuales y del estado actual antes del pulso de reloj. La última columna de la tabla consiste en tablas de excitación para cada tipo de flip-flop. Estas tablas definen el valor de la entrada o los valores necesarios para obtener cada uno de los valores posibles del próximo estado después del pulso de reloj, dando el valor de estados actual antes del pulso de reloj. Pueden emplearse las tablas de excitación para determinar las ecuaciones de entrada de los flip-flops a partir de la información de la tabla de estados.

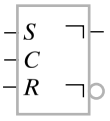
Históricamente, los flip-flops *JK* fueron una versión modificada de los flip-flops SR maestro-esclavo. Mientras el flip-flop *SR* produce salidas indefinidas y un comportamiento indeterminado para $S = R = 1$, el flip-flop *JK* genera por su salida el complemento de su valor actual. La versión del flip-flop *JK* maestro-esclavo tiene un comportamiento activo por pulso y, ade-

□ TABLA 6-7
Flip-flops, su diagrama lógico, tabla y ecuación característica y tabla de excitación

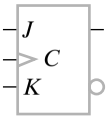
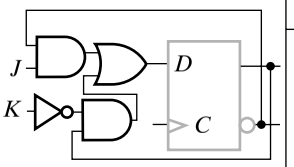
Tipo D

Símbolo	Diagrama lógico	Tabla de funcionamiento			Ecuación característica	Tabla de excitación		
	Ver Figura 6-13	***D***	***Q*(*t*+1)**	**Operación**	$Q(t+1) = D(t)$	***Q*(*t*+1)**	***D***	**Operación**
		0	0	Reset		0	0	Reset
		1	1	Set		1	1	Set

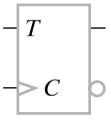
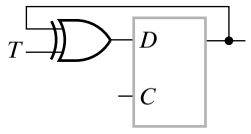
SR

Símbolo	Diagrama lógico	Tabla de funcionamiento				Ecuación característica	Tabla de excitación				
	Ver Figura 6-10	***S***	***R***	***Q*(*t*+1)**	**Operación**	$Q(t+1) = S(t) + \overline{R}(t)\cdot Q(t)$	***Q*(*t*)**	***Q*(*t*+1)**	***S***	***R***	**Operación**
		0	0	$Q(t)$	No cambia		0	0	0	*X*	No cambia
		0	1	0	Reset		0	1	1	0	Set
		1	0	1	Set		1	0	0	1	Reset
		1	1	?	Indefinido		1	1	*X*	0	No cambia

JK

Símbolo	Diagrama lógico	Tabla de funcionamiento				Ecuación característica	Tabla de excitación				
		***J***	***K***	**Q(t+1)**	**Operación**	$Q(t+1) = J(t)\cdot\overline{Q}(t)+\overline{K}(t)\cdot Q(t)$	***Q*(*t*)**	***Q*(*t*+1)**	***J***	***K***	**Operación**
		0	0	$Q(t)$	No cambia		0	0	0	*X*	No cambia
		0	1	0	Reset		0	1	1	*X*	Set
		1	0	1	Set		1	0	*X*	1	Reset
		1	1	$\overline{Q}(t)$	Complemento		1	1	*X*	0	No cambia

T

Símbolo	Diagrama lógico	Tabla de funcionamiento			Ecuación característica	Tabla de excitación		
		***T***	***Q*(*t*+1)**	**Operación**	$Q(t+1) = T(t) \oplus Q(t)$	***Q*(*t*+1)**	***T***	**Operación**
		0	$Q(t)$	No cambia		$Q(t)$	0	No cambia
		1	$\overline{Q}(t)$	Complemento		$\overline{Q}(t)$	1	Complemento

más, exhibe una propiedad llamada «captación de 1». Una vez que $J = 1$ o $K = 1$, de tal manera que el maestro cambie a su estado opuesto, el maestro no puede volver a su estado anterior antes que el pulso de reloj finalice, con independencia de los valores de J y K. Esto empeora el problema del tiempo de setup que ya existe para el flip-flop activo por pulso. Se aplica la misma solución que para los flip-flops *SR* (es decir, haciendo que el tiempo de setup, t_s, dure un pulso de disparo completo). Para evitar esta contribución adicional a la longitud del ciclo del reloj empleamos sólo flip-flops *JK* disparados por flanco construidos a partir de flip-flops *D* también disparados por flanco.

En la Tabla 6-7, se muestra el símbolo para un flip-flop *JK* disparado por flanco positivo así como su diagrama lógico empleando un flip-flop *D* activo por flanco positivo. La tabla característica dada describe el comportamiento del flip-flop *JK*. La entrada *J* se comporta como la entrada *S* para poner en set al flip-flop. La entrada *K* es similar a la entrada *R* para resetear el flip-flop. La única diferencia entre los flip-flops *SR* y los flip-flops *JK* es su respuesta cuando ambas entradas son iguales a 1. Como puede verificarse a partir del diagrama lógico, esta condición complementa el estado del flip-flop *JK*. Cuando $J = 1$ y $Q = 0$, entonces $D = 1$, complementando las salidas del flip-flop *JK*. Cuando $K = 1$ y $Q = 1$, entonces $D = 0$, complementando las salidas del flip-flop *JK*. Esto demuestra que, sin tener en cuenta el valor de Q, la condición $J = 1$ y $K = 1$ provoca la complementación de las salidas del flip-flop en contestación a un pulso de reloj. El comportamiento del siguiente estado se resume en la columna de la tabla característica de la Tabla 6-7. La entrada del reloj no se muestra explícitamente, pero se supone que ha ocurrido un pulso de reloj entre el estado actual y el próximo estado de Q.

El flip-flop *T* es equivalente al flip-flop *JK* con *J* y *K* unidos para que $J = K = T$. Con esta conexión, sólo se aplican las combinaciones $J = 0$, $K = 0$ y $J = 1$, $K = 1$. Si tomamos la ecuación característica para el flip-flop *JK* y hacemos esta conexión, la ecuación se vuelve

$$Q(t + 1) = T\bar{Q} + \bar{T}Q = T \oplus Q$$

El símbolo para el flip-flop *T* y su diagrama lógico están basados en la ecuación precedente que se da en la Tabla 6-7. La ecuación característica para el flip-flop *T* es simplemente la dada, y la tabla característica en la Tabla 6-7 muestra que para $T = 0$, las salidas del flip-flop *T* permanecen inalteradas, y para $T = 1$, las salidas se complementan. Puesto que el flip − flop *T* sólo puede mantener su estado inalterado o puede complementar su estado, no hay ninguna manera de establecer un estado inicial usando únicamente la entrada *T* sin añadir circuitería externa que muestree el valor actual de la salida. Así, el flip-flop *T* normalmente se inicializa a un estado conocido usando un set o un reset asíncrono.

6-7 REPRESENTACIÓN HDL PARA CIRCUITOS SECUENCIALES – VHDL

En el Capítulo 4 se empleó VHDL para describir circuitos combinacionales. Del mismo modo VHDL puede describir elementos de almacenamiento y circuitos secuenciales. En esta sección, como ejemplos de empleo de VHDL se ilustran las descripciones de un flip-flop D disparado por flanco positivo y de un circuito detector de secuencia. Estas descripciones suponen nuevos conceptos de VHDL, el más importante de ellos es el *proceso*. Hasta ahora, mediante sentencias concurrentes se han descrito combinaciones de condiciones y acciones en VHDL. Una sentencia concurrente, sin embargo, está limitada en cuanto a la complejidad que puede representar. Usualmente, los circuitos secuenciales a describir son bastante complejos y dicha descripción es muy difícil de plasmar con una sentencia concurrente. Un proceso puede verse como la alterna-

tiva a una sentencia concurrente con un poder descriptivo considerablemente mayor. Varios procesos se pueden ejecutar concurrentemente, y un proceso se puede ejecutar concurrentemente con las sentencias concurrentes.

Normalmente, el cuerpo de un proceso implementa un programa secuencial. Sin embargo, los valores de las señales que se asignan durante el proceso, sólo cambian cuando el proceso se completa. Si la porción de un proceso ejecutada es

```
B <= A;
C <= B;
```

entonces, cuando el proceso se completa, `B` contendrá los contenidos originales de `A` y `C` contendrá los contenidos originales de `B`. Por el contrario, después de la ejecución de estas dos sentencias en un programa, `C` contendría los contenidos originales de `A`. Para conseguir este comportamiento de programa, VHDL emplea otra estructura denominada *variable*. Frente a una señal que se evalúa tras un pequeño retardo, una variable se evalúa inmediatamente. Así, si `B` es una variable en la ejecución de

```
B := A;
C := B;
```

`B` se evaluará instantáneamente adquiriendo los contenidos de `A` y `C` se evaluará adquiriendo los nuevos contenidos de `B`, de modo que `C` contendrá finalmente los valores originales de la variable `A`. Las variables sólo aparecen dentro de los procesos. Véase el empleo de `:=` en lugar de `<=` para la asignación a variable.

EJEMPLO 6-5 VHDL para flip-flop *D* disparado por flanco positivo con reset

La estructura básica de un proceso se ilustra con el proceso del ejemplo de la Figura 6-31 que describe la arquitectura de un flip-flop *D* activo por flanco positivo. El proceso comienza con la palabra clave **`process`**. Opcionalmente, **`process`** puede ir precedido de un nombre de proceso seguido por dos puntos. A continuación, dentro del paréntesis hay dos señales, `CLK` y `RESET`. Esta es la *lista de sensibilidad* para el proceso. Si `CLK` o `RESET` cambian, entonces el proceso se ejecuta. En el general, un proceso se ejecuta siempre que cambie una señal o variable de su lista de sensibilidad. Es importante observar que la lista de sensibilidad no es una lista de parámetros que contiene todas las entradas y salidas. Por ejemplo, `D` no aparece, ya que un cambio en su valor no genera un cambio en el valor de `Q`. A continuación de la lista de sensibilidad, el proceso comienza con la palabra clave **`begin`**, y termina al final del proceso con la palabra clave **`end`**. La palabra **`process`** a continuación del **`end`** es opcional.

Dentro del cuerpo del proceso pueden aparecer estructuras condicionales adicionales de VHDL. En la Figura 6-31 es notable el ejemplo del **if-then-else**. La estructura general de un **if-then-else** en VHDL es

```
if condición then
   sucesión de sentencias
{elsif condicion then
   sucesión de sentencias}
else
   sucesión de sentencias
end if;
```

```
-- Flip-flop activo por flanco de subida con reset:
-- Descripción de proceso VHDL
library ieee;
use ieee.std_logic_1164.all;
entity dff is
   port(CLK, RESET, D : in std_logic;
        Q : out std_logic);
end dff;

architecture pet_pr of dff is
-- Implementa un flip-flop disparado por flanco de subida
-- con reset asíncrono.

begin
process (CLK, RESET)
   begin
     if (RESET = '1') then
       Q <= '0';
     elsif (CLK'event and CLK = '1') then
           Q <= D;
        end if;
     end if;
   end process;
end;
```

□ **FIGURA 6-31**
Descripción de un flip-flop disparado por flanco de subida con reset empleando un proceso VHDL

Las sentencias dentro de las llaves { } pueden aparecer desde ninguna hasta cualquier número de veces. El **if-then-else** dentro de un proceso es similar al efecto de la sentencia de asignación concurrente **when else**. Ilustrándolo, tenemos

```
if A = '1' then
   Q <= X;
elsif B = '0' then
   Q <= Y;
else
   Q <= Z;
end if;
```

Si A es 1, entonces Q del flip-flop se carga con los contenidos de X. Si A es 0 y B es 0, entonces Q toma los contenidos de Y. En cualquier otro caso, Q adquiere los contenidos de Z. El resultado final para las cuatro combinaciones de valores en A y B es

```
A = 0,  B = 0    Q <= Y
A = 0,  B = 1    Q <= Z
A = 1,  B = 0    Q <= X
A = 1,  B = 1    Q <= X
```

La ejecución de sentencias condicionales más complejas puede lograrse anidando estructuras **if-then-else**, como en el siguiente código:

```
if A = '1' then
   if C = '0' then
      Q <= W;
   else
      Q <= X;
  end if;
elsif B = '0' then
   Q <= Y;
else
   Q <= Z;
end if;
```

El resultado final para las ocho combinaciones de valores en *A*, *B* y el *C* es

```
A = 0,  B = 0,  C = 0    Q <= Y
A = 0,  B = 0,  C = 1    Q <= Y
A = 0,  B = 1,  C = 0    Q <= Z
A = 0,  B = 1,  C = 1    Q <= Z
A = 1,  B = 0,  C = 0    Q <= W
A = 1,  B = 0,  C = 1    Q <= X
A = 1,  B = 1,  C = 0    Q <= W
A = 1,  B = 1,  C = 1    Q <= X
```

Con la información presentada hasta ahora ya puede estudiarse el flip-flop *D* disparado por flanco positivo de la Figura 6-31. La lista de sensibilidad para el proceso incluye `CLK` y `RESET`, de modo que el proceso se ejecuta si `CLK`, `RESET` o ambas cambian de valor. En un flip-flop activo por flanco positivo, si el valor de `D` cambia, el valor de `Q` no cambia por lo que `D` no aparece en la lista de sensibilidad. Basado en el **if-then-else**, si `RESET` es 1, la salida `Q` se resetea a 0. Por otra parte, si el reloj cambia de valor, que se representa añadiendo `'event` a `CLK`, y el nuevo valor del reloj es 1, que se representa por `CLK='1'`, ha ocurrido un flanco positivo en `CLK`. El resultado de que se produzca un flanco positivo es la carga del valor `D` en el flip-flop para que aparezca en la salida. Observe que, debido a la estructura del **if-then-else**, la entrada `RESET` igual a 1 domina el comportamiento sincrono del flip-flop *D* provocando que la salida `Q` pase a 0. Pueden emplearse sencillas descripciones similares para representar otros tipos de flip-flops y mecanismos de disparo. ■

EJEMPLO 6-6 VHDL para el detector de secuencia

Un ejemplo más complejo se presenta en las Figuras 6-32 y 6-33 que representan el detector de secuencia del diagrama de estados de la Figura 6-24(d). La arquitectura en esta descripción consiste en tres procesos distintos que se pueden ejecutar simultáneamente y pueden interactuar a través de señales compartidas. Se introducen nuevos conceptos como son las declaraciones de tipo para definir los nuevos tipos y sentencias *case* para manejar las condiciones.

```
-- Detector de secuencia. Descripción de proceso VHDL
-- (véase la Figura 6-24(d) para el diagrama de estados)
library ieee;
use ieee.std_logic_1164.all;
entity seq_rec is
   port(CLK, RESET, X: in std_logic;
        Z: out std_logic);
end seq_rec;

architecture process_3 of seq_rec is
   type state_type is (A, B, C, D);
   signal state, next_state : state_type;
begin

-- Process 1 - state_register: implementa un almacenamiento disparado
-- por flanco de subida con reset asíncrono.
   state_register: process (CLK, RESET)
   begin
     if (RESET = '1') then
        state <= A;
     elsif (CLK'event and CLK = '1') then
           state <= next_state;
        end if;
     end if;
end process;

-- Process 2 - función del estado futuro: implementa el próximo
-- estado como una función de X y state.
   next_state_func: process (X, state)
   begin
      case state is
         when A =>
      if X = '1' then
         next_state <= B;
      else
         next_state <= A;
            end if;
        when B =>
      if X = '1' then
         next_state <= C;
      else
         next_state <= A;
            end if;
```

□ **FIGURA 6-32**
Descripción VHDL de proceso de un detector de secuencia

La declaración de tipo permite definir nuevos tipos de manera similar a los tipos existentes como el `std_logic`. Una declaración de tipo empieza con la palabra clave **`type`** seguida por el nombre del nuevo tipo, la palabra clave **`is`**, y, dentro de paréntesis, la lista de valores para las señales del nuevo tipo. Usando el ejemplo de la Figura 6-31, tenemos

```
type state_type is (A, B, C, D);
```

```
-- Detector de secuencia. Descripción de proceso VHDL (continuación)
  when C =>
     if X = '1' then
        next_state <= C;
     else
        next_state <= D;
           end if;
        when D =>
     if X = '1' then
        next_state <= B;
     else
        next_state <= A;
           end if;
      end case;
   end process;

-- Process 3 - función de salida: implementa la salida como
-- una función de X y state.
   output_func: process (X, state)
   begin
      case state is
         when A =>
       Z <= '0';
    when B =>
       Z <= '0';
    when C =>
       Z <= '0';
          when D =>
     if X = '1' then
        Z <= '1';
     else
        Z <= '0';
           end if;
      end case;
   end process;
end;
```

□ **FIGURA 6-33**
Descripción VHDL de un detector de secuencia (continuación)

El nombre del nuevo tipo es `state_type` y los valores en este caso son los nombres de los estados de la Figura 6-24(d). Una vez que se ha declarado un **`type`**, puede usarse para declarar señales o variables. En el ejemplo de la Figura 6-31,

```
signal state, next_state : state_type;
```

indica que `state` y `next_state` son señales del tipo `state_type`. Así, `state` y `next_state` pueden tener los valores `A`, `B`, `C` y `D`.

El **if-then-else** básico (sin usar el **`elsif`**) toma una bifurcación basada en si una condición es VERDADERA o FALSA. Sin embargo, la sentencia **case** puede tomar múltiples decisiones

basándose en que un número de sentencias sean VERDADERAS. Una forma simplificada para la declaración genérica **case** es

```
case expresión is
   {when opciones =>
      sucesión de sentencias;}
end case;
```

Las opciones deben ser valores asignables a una señal del tipo empleado en la expresión. La declaración **case** tiene un efecto similar a la sentencia de asignación concurrente **with-select**.

En el ejemplo de las Figuras 6-32 y 6-33, `Process 2` emplea una sentencia **case** para definir el siguiente estado del detector de secuencia. La declaración **case** toma una decisión múltiple basada en el estado actual del circuito: `A`, `B`, `C` o `D`. La sentencia **if-then-else** se emplea dentro de cada una de las alternativas para tomar una decisión binaria basada en si la entrada `X` es 1 o 0. Entonces, se emplean sentencias de asignación concurrentes para asignar el próximo estado en función de las ocho posibles combinaciones de valores de estados y valores de entrada. Por ejemplo, considere la alternativa **when** `B`. Si `X` es igual a 1, entonces el próximo estado será `C`; si `X` es igual a 0, entonces el próximo estado será `A`. Esto se corresponde con las dos transiciones del estado `B` de la Figura 6-24(d). Para circuitos más complejos, también pueden emplearse sentencias **case** para manejar las condiciones de la entrada.

Con esta breve introducción a la sentencia **case** ya podemos estudiar detectores de secuencia globales. Cada uno de los tres procesos tiene una función distinta, pero los procesos interactúan para proporcionar el detector de secuencia global. `Process 1` describe el almacenamiento del estado. Observe que la descripción está realizada con flip-flops disparados por flanco positivo. Sin embargo, hay dos diferencias. Las señales involucradas son del tipo `state_type` en lugar del tipo `std_logic`. Segundo, el estado que resulta de aplicar la señal `RESET` es el estado `A` en lugar del estado 0. También, puesto que estamos usando nombres de estados como `A`, `B` y `C`, el número de variables de estados (es decir, el número de flip-flops) no está especificado y desconocemos los códigos de estados. `Process 1` es el único de los tres procesos que contiene almacenamiento.

`Process 2` describe la función siguiente estado, discutida anteriormente. En este caso, la lista de sensibilidad contiene la señal `X` y los estados. En el general, para describir lógica combinacional, deben aparecer todas las entradas en la lista de sensibilidad, ya que siempre que una entrada cambie, el proceso tendrá que ejecutarse.

`Process 3` describe la función de salida. Se emplea la misma sentencia **case** que en el `Process 2` con el estado como expresión de decisión. En lugar de asignar nombres de estados al siguiente estado, se asignan a `Z` valores 0 o 1. Si el valor que se asigna es el mismo independientemente del valor de `X`, no se necesita ningún **if-then-else**, por lo que sólo aparece un **if-then-else** para el estado `D`. Si hay variables de entrada múltiples, como anteriormente, se emplearán combinaciones anidadas de **if-then-else** o una sentencia **case** para representar las condiciones de las salidas en función de las entradas. Este ejemplo es un Diagrama de estados Mealy en el que la salida es una función de las entradas del circuito. Si fuera un Diagrama de Moore, con la salida dependiente sólo del estado, la entrada `X` no aparecería en la lista de sensibilidad, y no habría ninguna estructura **if-then-else** en la sentencia **case**. ■

Hay un peligro habitual cuando se emplea un **if-then-else** o un **case**. Durante la síntesis pueden aparecer elementos de almacenamiento indeseados en forma de latches o flip-flops. Para el sencillo **if-then-else** utilizado en la Figura 6-31, este peligro se manifiesta obteniendo una especificación que se sintetiza en un flip-flop. Además de las dos señales de entrada, `RESET` y `CLK`,

se añade la señal `CLK'event`, que se obtiene aplicando el atributo predefinido *event* a la señal `CLK`. `CLK'event` es `VERDADERO` si el valor de `CLK` cambia. Todas las posibles combinaciones de estas señales se representan en la Tabla 6-8. Siempre que `RESET` es 0 y `CLK` está fijo a 0 o a 1 o tiene un flanco negativo, no se especifica ninguna acción. En VHDL, se supone que, para cualquier combinación de condiciones para la que no se especifica ninguna acción en las sentencias

□ **TABLA 6-8**
Generación de almacenamiento en VHDL

Entradas			Acción
`RESET = 1`	**`CLK = 1`**	**`CLK'event`**	
FALSO	FALSO	FALSO	Sin especificar
FALSO	FALSO	VERDADERO	Sin especificar
FALSO	VERDADERO	FALSO	Sin especificar
FALSO	VERDADERO	VERDADERO	`Q <= D`
VERDADERO	—	—	`Q <= '0'`

if-then-else o **case,** el lado izquierdo de la sentencia de asignación permanece inalterado. Esto es equivalente a `Q <= Q`, originando un almacenamiento. Entonces, todas las combinaciones de condiciones deben tener como resultado una acción específica cuando no exista almacenamiento. Si esta no es una situación natural, puede usarse **`others`** en el **if-then-else** o en el **case**. Si hay valores binarios en la sentencia **case**, como en la Sección 4-7, debe recurrirse a **others** para manejar las combinaciones incluyendo los siete valores además del 0 y el 1 permitidos para std_logic.

Juntos, los tres procesos empleados para el detector de secuencia describen el almacenamiento del estado, la función del estado futuro y la función de salida para un circuito secuencial. Puesto que estos son todos los componentes del circuito secuencial a nivel del diagrama de estados, la descripción está completa. El empleo de tres procesos distintos para describir el circuito secuencial es solo una metodología. Pueden combinarse dos procesos o los tres para obtener descripciones más elegantes. No obstante, la descripción de tres procesos es más fácil para los principiantes en VHDL y además funciona bien con las herramientas de síntesis.

Para sintetizar el circuito en lógica real se necesita una asignación de estados además de una librería tecnológica. Muchas herramientas de síntesis harán la asignación de estados independientemente o en base a una directiva del usuario. El usuario también puede especificar explícitamente la asignación de estados. Esto puede hacerse en VHDL empleando un tipo enumerado. La codificación para el diagrama de estados de las Figuras 6-32 y 6-33 se especifica añadiendo lo siguiente después de la declaración del **`type`** `state_type`:

```
attribute enum_encoding: string;
attribute enum_encoding of state_type:
type is "00, 01, 10, 11";
```

Esto no es una construcción normal VHDL, pero es reconocida por muchas herramientas de síntesis. Otra opción es no usar una declaración de tipo para los estados, sino declarar las variables de estados como señales y usar los códigos reales para los estados. En este caso, si los estados aparecen en la salida de la simulación, aparecerán como estados codificados.

6-8 REPRESENTACIÓN DE HDL PARA CIRCUITOS SECUENCIALES-VERILOG

En el Capítulo 4, se empleó Verilog para describir circuitos combinacionales. Igualmente, Verilog puede describir elementos de almacenamiento y circuitos secuenciales. En esta sección, para ilustrar estos usos de Verilog recurriremos a describir un flip-flop *D* disparado por flanco positivo y un circuito **detector de secuencia**. Estas descripciones implicarán nuevos conceptos de Verilog, los más importantes son: el proceso y el tipo registro.

Hasta ahora se han empleado sentencias de asignación para describir combinaciones de condiciones y acciones en Verilog. Sin embargo, una sentencia de asignación continua está limitada en cuanto a lo que puede describir. Un proceso puede verse como una opción frente a la sentencia de asignación continua, con un poder descriptivo considerablemente mayor. Es posible ejecutar concurrentemente varios procesos y un proceso se puede ejecutar concurrentemente con sentencias de asignación continuas.

Dentro de un proceso, se emplean sentencias de asignación procedimentales que no son asignaciones continuas. Es por ello que los valores asignados deben ser almacenados en el tiempo. Esta retención de información se logra gracias al uso del tipo registro **`reg`** en lugar del tipo **`wire`** para los nodos. La palabra clave para el tipo registro es **`reg`**. Observe que aunque un nodo sea de tipo registro esto no significa que un registro real esté asociado con su implementación. Hay otras condiciones adicionales que se necesitan para que exista un registro real.

Existen dos tipos básicos de procesos, el proceso **`initial`** y el proceso **`always`**. El proceso **`initial`** sólo se ejecuta una vez, comenzando en $t = 0$. El proceso **`always`** también se ejecuta para $t = 0$, pero se ejecuta repetidamente después de $t = 0$. Para prevenir la ejecución desenfrenada, descontrolada, se necesita algún control de tiempos en forma de retardo o tiempo de espera basado en algún evento. El operador # seguido por un entero se emplea para especificar el retardo. El operador **@** puede verse como «espera al evento». Una expresión a continuación de @ describe el evento o eventos cuya ocurrencia provocará que el proceso se ejecute.

El cuerpo de un proceso es como un programa secuencial. El proceso comienza con la palabra clave **`begin`** y acaba con la palabra clave **`end`**. Hay sentencias de asignación procedimentales que constituyen el cuerpo del proceso. Estas sentencias de asignación son clasificadas como *blocking* o *nonblocking*. Las asignaciones *blocking* usan = como operador de la asignación y las *nonblocking* usan <= como operador. Las *asignaciones blocking* se ejecutan secuencialmente, como un programa en un lenguaje procedimental como C. Las asignaciones *nonblocking* evalúan el lado derecho, pero no efectúan la asignación hasta que se hayan evaluado todos los lados derechos. Las asignaciones pueden ilustrarse por el siguiente cuerpo del proceso en el que `A`, `B` y `C` son de tipo **`reg`**:

```
begin
   B = A;
   C = B;
end
```

La primera sentencia transfiere los contenidos de `A` a `B`. La segunda sentencia transfiere los nuevos contenidos de `B` a `C`. A la finalización del proceso, `C` contiene los contenidos originales de `A`.

Suponga que el mismo cuerpo del proceso usa asignaciones *nonblocking*:

```
begin
   B <= A;
   C <= B;
end
```

La primera sentencia transfiere los contenidos de A a B y la segunda sentencia transfiere los contenidos originales de B a C. A la finalización del proceso, C contiene los contenidos originales de B, no los de A. Efectivamente, la dos sentencias se han ejecutado concurrentemente en lugar de en serie. Se emplean asignaciones *nonblocking*, excepto en los casos en los que queremos registros (tipo **reg**) que se evalúen secuencialmente.

EJEMPLO 6-7 Verilog para flip-flop *D* con reset activo por flanco positivo

Estos nuevos conceptos pueden aplicarse ahora a la descripción verilog de un flip-flop *D* disparado por flanco positivo dada en la Figura 6-34. Se declara el módulo y sus entradas y salidas. Q se declara como tipo **reg** puesto que guardará la información. El proceso comienza con la palabra clave **always**. Lo siguiente es @(**posedge** CLK **or posedge** RESET). Ésta es la declaración del *evento de control* del proceso que inicia la ejecución del proceso si ocurre un evento (es decir, si ocurre un cambio determinado en una señal determinada). Para el flip-flop *D*, tanto si CLK o RESET cambian a 1, entonces el proceso se ejecuta. Es importante tener en cuenta que la declaración de eventos de control no es una lista de parámetros que contiene todas las entradas. Por ejemplo, la entrada D no aparece, ya que un cambio en su valor no origina un cambio en el valor de Q. Tras la declaración de eventos de control, el proceso comienza con la palabra clave **begin**, y finaliza con la palabra clave **end**.

Dentro del cuerpo del proceso, pueden aparecer otras estructuras condicionales adicionales de Verilog. Es interesante, en el ejemplo de la Figura 6-34, **if-else**. La estructura general de un **if-else** en Verilog es

```
if (condición)
   begin declaraciones procedimentales end
{else if (condición)
   begin declaraciones procedimentales end}
{else
   begin declaraciones procedimentales end}
```

```
// Flip-flop activado por flanco de subida con reset:
// descripción de proceso Verilog

module dff_v(CLK, RESET, D, Q);
   input CLK, RESET, D;
   output Q;
   reg Q;

always @(posedge CLK or posedge RESET)
begin
   if (RESET)
      Q <= 0;
   else
      Q <= D;
end
endmodule
```

□ **FIGURA 6-34**
Descripción Verilog de proceso de un flip-flop disparado por flanco de subida con reset

Si hay una sola declaración procedimental, entonces el **begin** y **end** son innecesarios:

```
if (A == 1)
   Q <= X;
else if (el B == 0)
   Q <= SI;
else
   Q <= Z;
```

Observe que en las condiciones se emplea un doble igual entre las señales a comparar. Si A es 1, entonces la Q del flip-flop se carga con el contenido de X. Si A es 0 y B es 0, entonces la Q del flip-flop se carga con el contenido de Y. En cualquier otro caso, la Q se cargará con los contenidos de Z. El resultado final para las cuatro combinaciones de valores en A y en B es

```
A = 0,  B = 0   Q <= Y
A = 0,  B = 1   Q <= Z
A = 1,  B = 0   Q <= X
A = 1,  B = 1   Q <= X
```

En efecto, el **if-else** dentro de un proceso es similar al operador condicional de la sentencia de asignación continua introducida con anterioridad. El operador condicional puede usarse dentro de un proceso, pero el **if-else** no puede emplearse en una sentencia de asignación continua.

Pueden conseguirse unas construcciones más complejas anidando estructuras del tipo **if-else**. Por ejemplo, podríamos tener

```
if (A == 1)
   if (C == 0)
      Q <= W;
   else
      Q <= X;
else if (el B == 0)
   Q <= Y;
   else
   Q <= Z;
```

En este tipo de estructura, un **else** se asocia con el **if** anterior más próximo que no tenga todavía un **else**. El resultado final para las ocho combinaciones de valores en *A*, *B* y *C* es

```
A = 0,  B = 0,  C = 0   Q < = Y
A = 0,  B = 0,  C = 1   Q < = Y
A = 0,  B = 1,  C = 0   Q < = Z
A = 0,  B = 1,  C = 1   Q < = Z
A = 1,  B = 0,  C = 0   Q < = W
A = 1,  B = 0,  C = 1   Q < = X
A = 1,  B = 1,  C = 0   Q < = W
A = 1,  B = 1,  C = 1   Q < = X
```

Volviendo al **if-else** del flip-flop *D* disparado por flanco positivo de la Figura 6-34, suponiendo que se ha producido un flanco en CLK o en RESET, si RESET es 1, la salida Q del flip-flop se resetea a 0. En cualquier otro caso, el valor en D se almacena en el flip-flop de modo que Q se hace igual a D. Debido a la estructura del **if-else**, RESET igual a 1 domina el comportamiento sincrono del flip-flop *D*, lo que provoca que la salida Q sea 0. Descripciones igual de sencillas pueden utilizarse para representar otros tipos de flip-flops y otros tipos de disparo. ■

EJEMPLO 6-8 Verilog para el detector de secuencia

En la Figura 6-35 se ilustra un ejemplo más complejo que representa el diagrama de estados del detector de secuencia de la Figura 6-24(d). En esta descripción la arquitectura consiste en tres procesos distintos que se pueden ejecutar simultáneamente y que interactúan mediante señales. Se incluyen nuevos conceptos como son la codificación de estados y la sentencia **case** para manejar las condiciones.

```
// Detector de secuencia: descripción de proceso Verilog
// (véase la Figura 6-24(d) para el diagrama de estados)
module seq_rec_v(CLK, RESET, X, Z);
  input CLK, RESET, X;
  output Z;
  reg [1:0] state, next_state;
  parameter A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;
  reg Z;
// registro de estado: almacenamiento disparado por flanco de subida
// con reset asíncrono.
always @(posedge CLK or posedge RESET)
begin
  if (RESET == 1)
    state <= A;
  else
    state <= next_state;
end
// función estado futuro: las variables son X y state
always @(X or state)
begin
  case (state)
    A:   if (X == 1)
           next_state <= B;
         else
           next_state <= A;
    B:   if(X) next_state <= C;else next_state <= A;
    C:   if(X) next_state <= C;else next_state <= D;
    D:   if(X) next_state <= B;else next_state <= A;
  endcase
end
// función de salida: las variables son X y state
always @(X or state)
begin
  case (state)
    A: Z <= 0;
    B: Z <= 0;
    C: Z <= 0;
    D: Z <= X ? 1 : 0;
   endcase
end
endmodule
```

□ **FIGURA 6-35**
Descripción de proceso Verilog del detector de secuencias

En la Figura 6-35, se declaran el módulo `seq_rec_v` y las variables de entrada y salida `CLK`, `RESET`, `X` y `Z`. A continuación, se declaran los registros `state` y `next_state`. Véase cómo, ya que `next_state` no necesita almacenarse, también podría declararse como un **wire**, pero, puesto que se asigna dentro de un **`always`**, debe declararse como un **`reg`**. Ambos registros son de dos bits, con el bit más significativo (MSB) numerado como 1 y el bit menos significativo (LSB) numerado como 0.

A continuación, se da un nombre a cada uno de los estados de `state` y `next_state`, y se asignan los códigos binarios para dichos nombres. Esto puede hacerse mediante una declaración de parámetros o mediante una directiva **`define`** del compilador. Nosotros emplearemos la declaración de parámetros, ya que la directiva del compilador requiere un inoportuno `'` antes de cada estado a lo largo de la descripción. A partir del diagrama de la Figura 6-24(d), los estados que se obtienen son *A, B, C* y *D*. Además, las declaraciones de parámetros permiten asignar, a la vez, los códigos de estados a cada uno de dichos estados. La notación utilizada para definir los códigos de estado es `2'b` seguido por el código binario. El `2` indica que hay dos bits en el código y la `'b` indica que la base del código a emplear es binaria.

El **if-else** (sin usar el **else if**) toma una bifurcación basándose en si una condición es VERDADERA o FALSA. Por contra, la sentencia **case** puede tomar decisiones múltiples en función de que una de varias sentencias sea VERDADERA. Una forma simplificada para la declaración genérica **case** es

```
case expresión
   {case expresión: sentencias}
endcase
```

en la que las llaves{ } representan una o más entradas.

La `expresión del case` debe tener valores que pueden asignarse al tipo de señal empleado en la `expresión`. Normalmente, son secuencias de sentencias múltiples. En el ejemplo de la Figura 6-35, la declaración del **case** para la función del estado futuro toma una decisión de entre muchas en base al estado actual del circuito, `A`, `B`, `C` o `D`. En cada expresión del case, se recurre a sentencias condicionales de varios tipos para tomar una decisión binaria basándonos en si la entrada `X` es 1 o 0. Sentencias de asignación *nonblocking* se emplean para asignar el siguiente estado basado en las ocho combinaciones posibles del valor del estado y del valor de la entrada. Por ejemplo, considere la expresión `B`. Si `X` es igual a 1, entonces el siguiente estado será `C`; si `X` es igual a 0, entonces el siguiente estado será `A`. Esto se corresponde con las dos transiciones desde el estado `B` en la Figura 6-24(d).

Con esta breve introducción a la sentencia **case** ahora ya podemos entender el detector de secuencia global. Cada uno de los tres procesos tiene una función distinta, pero los procesos interactúan para obtener el detector global de secuencia. El primer proceso describe el registro de estado para almacenar el estado del detector de secuencia. Observe que la descripción es similar a la de un flip-flop disparado por flanco positivo. Existen, sin embargo, dos diferencias. Primero, el registro de estado tiene dos bits. Segundo, el estado que resulta de aplicar `RESET` es el estado A en lugar del estado 0. El primer proceso es el único de los tres procesos que lleva implícito un almacenamiento.

El segundo proceso describe la función del estado futuro, como se discutió anteriormente. La declaración de eventos de control contiene las señales `X` y los `estados`. En general, para describir la lógica combinacional, todas las entradas han de estar presentes en la declaración de control de eventos, puesto que siempre que una entrada cambie, el proceso debería ejecutarse.

El último proceso describe la función de salida y emplea el mismo cuerpo de sentencias **case** que el proceso de la función estado futuro. En lugar de asignar los nombres de estado a `Z`,

se asignan los valores 0 y 1. Si el valor asignado es el mismo tanto para el valor 0 como para el valor 1 de X, no se necesita ninguna sentencia condicional y únicamente aparece la sentencia condicional para el estado D. Si hay múltiples variables de entrada, como se ilustró antes, puede emplearse una combinación más compleja de **if-else** para representar las condiciones de las salidas sobre las entradas. Este ejemplo es un Diagrama de Mealy en que la salida es función de las entradas del circuito. Si fuera un Diagrama de Moore, con la salida dependiente únicamente del estado, la entrada X no aparecería en la declaración de eventos de control y no habría ninguna estructura condicional dentro de la declaración del **case**. ■

Hay un peligro habitual cuando se emplea un **if-else** o un **case**. Durante la síntesis pueden aparecer elementos de almacenamiento indeseados en forma de latches o flip-flops. Para el sencillo **if-else** utilizado en la Figura 6-34, este peligro se manifiesta obteniendo una especificación que se sintetiza en un flip-flop. Además de las dos señales de entrada, RESET y CLK, se añaden los eventos **posedge** CLK y **posedge** RESET que son VERDADEROS si el valor de la señal correspondiente cambia de 0 a 1. En la Tabla 6-9 se muestran algunas combinaciones de estas señales. Cuando RESET no tiene un flanco de subida o RESET es 0 y CLK está fijo a 0 o a 1 o tiene un flanco de bajada no se especifica ninguna acción. En Verilog se supone que, para cualquier combinación de condiciones para la que no se especifica ninguna acción en las sentencias **if-else** o **case,** el lado izquierdo de la sentencia de asignación permanece inalterado. Esto es equivalente a Q <= Q, originando un almacenamiento. Entonces, todas las combinaciones de condiciones deben tener como resultado una acción específica cuando no exista almacenamiento. Para evitar la aparición de latches o flip-flops indeseados en un **if-else** hay que tener cuidado de incluir un **else** siempre que no se desee que se genere almacenamiento. Para la construcción **case** se debe incluir una sentencia **default** que defina qué debe hacerse para todos los casos no declarados explícitamente. En el ejemplo podría ponerse un sentencia **default** que especificase que el estado futuro es *A*.

Juntos, los tres procesos empleados para el detector de secuencia describen el almacenamiento de estados, la función de estado futuro y la función de salida para el circuito secuencial. Dado que éstos son todos los componentes de un circuito secuencial a nivel del diagrama de estados, la descripción está completa. La utilización de tres procesos distintos es sólo una metodología para la descripción del circuito secuencial. Por ejemplo, podrían combinarse fácilmente los procesos de estado futuro y de la salida. No obstante, la descripción de tres procesos es más fácil para los usuarios noveles de Verilog y funciona bien con las herramientas de síntesis.

□ **TABLA 6-9**
Generación de almacenamiento en Verilog

Entradas		Acción
posedge RESET y RESET = 1	**posedge CLK**	
FALSO	FALSO	Sin especificar
FALSO	VERDADERO	Q <= D
VERDADERO	FALSO	Q <= 0
VERDADERO	VERDADERO	Q <= 0

6–9 RESUMEN DEL CAPÍTULO

Los circuitos secuenciales son la base fundamental en la que se asientan la mayoría de los diseños digitales. Los flip-flops son los elementos de almacenamiento básicos para los circuitos secuenciales síncronos. Los flip-flops se construyen a partir de elementos más fundamentales llamados latches. Por ellos mismos los latches son transparentes y, en consecuencia, son muy difíciles de usar en circuitos secuenciales síncronos que emplean un solo reloj. Cuando se combinan los latches para formar los flip-flops, se logran elementos de almacenamiento no transparentes muy convenientes para utilizarlos en dichos circuitos. Los flip-flops tienen dos métodos de disparo: el maestro-esclavo y activo por flanco. Además, hay varios tipos de flip-flops, incluyendo el *D*, *SR*, *JK* y *T*.

Los circuitos secuenciales se obtienen del empleo de estos flip-flops y lógica combinacional. Los circuitos secuenciales se pueden analizar para obtener las tablas de estados y los diagramas de estados que plasman el comportamiento de los circuitos. También puede realizarse el análisis mediante el empleo de la simulación lógica.

Estos mismos diagramas y tablas de estado se pueden formular a partir de las especificaciones verbales de los circuitos digitales. Asignando códigos binarios a los estados y encontrando las ecuaciones de entrada de los flip-flops se pueden diseñar los circuitos secuenciales. El proceso del diseño incluye también problemas como encontrar la lógica para las salidas del circuito, resetear el estado en el encendido y controlar el comportamiento del circuito cuando entra en estados no usados por la especificación original. Finalmente, la simulación lógica juega un papel importante verificando que el circuito diseñado se ajusta a la especificación original.

Como alternativa al empleo de diagramas lógicos, de diagramas y tablas de estados, los circuitos secuenciales se pueden definir mediante descripciones en VHDL o Verilog. Estas descripciones, típicamente a nivel de comportamiento, proporcionan una aproximación potente y flexible a la especificación del circuito secuencial, tanto para la simulación como para la síntesis automática del circuito. Estas representaciones implican a los *procesos*, que proporcionan un poder descriptivo mayor que las sentencias de asignación concurrentes de VHDL y las sentencias de asignación continua de Verilog. Los procesos, que permiten la codificación del comportamiento de los circuitos de forma similar a un lenguajes de programación, usan las sentencias condicionales **if-then-else** y **case**, que pueden también emplearse para modelar eficazmente lógica combinacional.

REFERENCIAS

1. MANO, M. M.: *Digital Design*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2002.
2. ROTH, C. H.: *Fundamentals of Logic Design*, 4th ed. St. Paul: West, 1992.
3. WAKERLY, J. F.: *Digital Design: Principles and Practices*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2000.
4. *IEEE Standard VHDL Language Reference Manual*. (ANSI/IEEE Std 1076-1993; revision of IEEE Std 1076-1987). New York: The Institute of Electrical and Electronics Engineers, 1994.
5. PELLERIN, D. and D. TAYLOR: *VHDL Made Easy!* Upper Saddle River, NJ: Prentice Hall PTR, 1997.
6. STEFAN, S. and L. LINDH: *VHDL for Designers*. London: Prentice Hall Europe, 1997.

7. *IEEE Standard Description Language Based on the Verilog(TM) Hardware Description Language* (IEEE Std 1364-1995). New York: The Institute of Electrical and Electronics Engineers, 1995.

8. PALNITKAR, S.: *Verilog HDL: A Guide to Digital Design and Synthesis*. Upper Saddle River, NJ: SunSoft Press (A Prentice Hall Title), 1996.

9. CILETTI, M.: *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL*, Upper Saddle River, NJ: Prentice Hall, 1999.

10. THOMAS, D. E., and P. R. MOORBY: *The Verilog Hardware Description Language* 4th ed. Boston: Kluwer Academic Publishers, 1998.

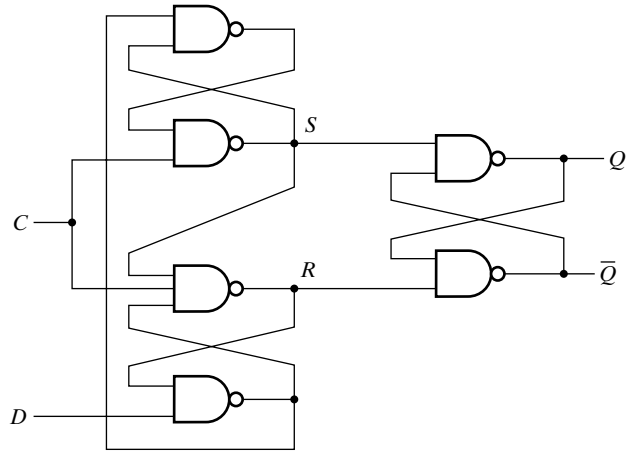
PROBLEMAS

El símbolo (+) indica problemas más avanzados y el asterisco (*) indica que la solución se puede encontrar en el sitio web del libro: http://www.librosite.net/Mano.

6-1. Realice una simulación lógica manual o por computadora similar a la mostrada en el Figura 6-5 para el latch $\overline{S}\overline{R}$ de la Figura 6-6. Construya la secuencia de entrada, teniendo presente que los cambios en el estado para este tipo de latch ocurren en respuesta a 0 en lugar de a 1.

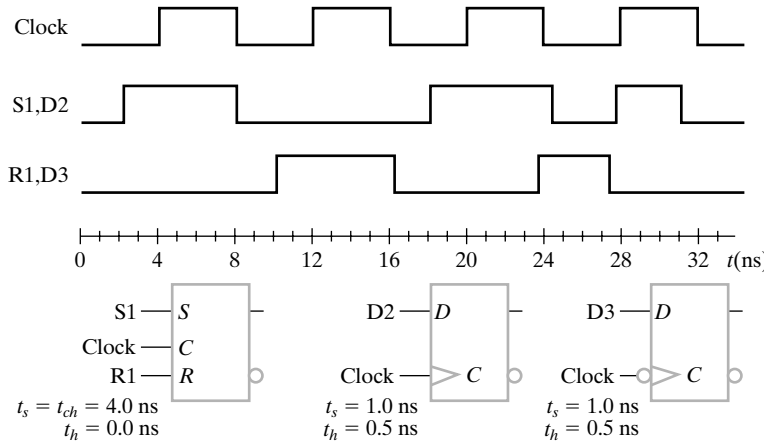
6-2. Realice una simulación lógica manual o por computadora similar a la dada en la Figura 6-5 para el latch *SR* con entrada de control *C* de la Figura 6-7. En particular, examine el comportamiento del circuito cuando *S* y *R* cambian mientras que *C* tiene el valor 1.

6-3. En la Figura 6-36 se muestra un diseño alternativo muy habitual para un flip-flop *D* disparado por flanco positivo. Simule el circuito manual o automáticamente para determinar si su comportamiento funcional es idéntico al del circuito de la Figura 6-13.



□ **FIGURA 6-36**
Circuito para el Problema 6-3

6-4. En la Figura 6-37 se muestran los cronogramas de las señales aplicadas a los flip-flops *SR* y *D*. Junto a los cronogramas de estos flip-flop se muestran los valores de sus parámetros temporales.



□ **FIGURA 6-37**
Cronograma y flip-flops para el Problema 6-4

(a) Indique los instantes del cronograma en los cuales hay violaciones en las combinaciones de entrada o en los parámetros temporales para la señal S1 del flip-flop 1.
(b) Indique las posiciones del cronograma en las cuales hay violaciones en las combinaciones de entrada o en los parámetros temporales para la señal R1 del flip-flop 1.
(c) Enumere los tiempos en los cuales hay violaciones de los parámetros temporales en la señal D2 del flip-flop 2.
(d) Enumere los tiempos en los cuales hay violaciones de los parámetros temporales en la señal D3 del flip-flop 3.

Se deben indicar las violaciones aunque el estado del flip-flop sea tal que estas violaciones no afecten al siguiente estado.

6-5. Las siguientes ecuaciones de entrada especifican un circuito secuencial con dos flip-flop, *A* y *B*, de tipo *D*, dos entradas, *X* e *Y*, y una salida *Z*:

$$D_A = \bar{X}A + XY \qquad D_B = \bar{X}A + XB \qquad Z = XB$$

(a) Dibuje el diagrama lógico del circuito.
(b) Obtenga la tabla de estados para el circuito.
(c) Obtenga el diagrama de estados.

6-6. *Un circuito secuencial tiene tres flip-flop *D* nombrados *A*, *B* y *C*, y una entrada *X*. El circuito se describe mediante las siguientes ecuaciones de entrada:

$$D_A = (B\bar{C} + \bar{B}C)X + (BC + \bar{B}\bar{C})\bar{X}$$

$$D_B = A$$

$$D_C = B$$

(a) Obtenga la tabla de estados para el circuito.
(b) Dibuje dos diagramas de estado, uno para $X = 0$ y el otro para $X = 1$.

6-7. Un circuito secuencial tiene un flip-flop *Q*, dos entradas *X* e *Y*, y una salida *S*. El circuito consiste en un flip-flop tipo *D* con una salida *S* y lógica adicional que implementa la función:

$$D = X \oplus Y \oplus S$$

con *D* como la entrada al flip-flop *D*. Obtenga la tabla y el diagrama de estados del circuito secuencial.

6-8. A partir del estado 00 del diagrama de estados de la Figura 6-19(a), determine las transiciones de estado y la secuencia de salida que se obtiene cuando se aplica la secuencia de entrada 10011011110.

6-9. Dibuje el diagrama de estados del circuito secuencial especificado por la tabla de estados en la Tabla 6-10.

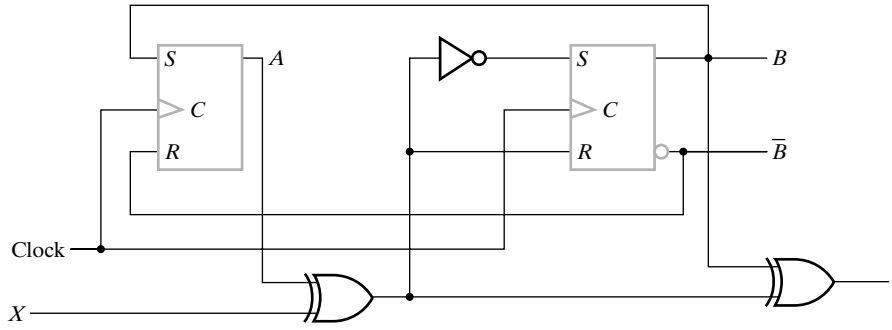
□ **TABLA 6-10**
Tabla de estados para el circuito del Problema 6-9

Estado actual		**Entradas**		**Estado futuro**		**Salida**
A	***B***	***X***	***Y***	***A***	***B***	***Z***
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	1
0	0	1	1	1	1	1
0	1	0	0	0	1	1
0	1	0	1	1	0	1
0	1	1	0	1	0	0
0	1	1	1	0	0	0
1	0	0	0	1	1	1
1	0	0	1	1	1	0
1	0	1	0	1	1	1
1	0	1	1	1	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	1
1	1	1	0	0	0	0
1	1	1	1	0	1	1

6-10. *Un circuito secuencial tiene dos flip-flop *SR*, una entrada *X*, y una salida *Y*. El diagrama lógico del circuito se muestra en la Figura 6-38. Obtenga la tabla y el diagrama de estados del circuito.

6-11. En la Figura 6-38 se muestra un circuito secuencial. Los parámetros temporales para las puertas y los flip-flop son los siguientes:

Inversor: $t_{pd} = 0.5$ ns
Puerta XOR: $t_{pd} = 2.0$ ns
Flip-flop: $t_{pd} = 2.0$ ns, $t_s = 1.0$ ns y $t_h = 0.25$ ns



□ **FIGURA 6-38**
Circuito para los Problemas 6-10, 6-11 y 6-12

(a) Encuentre el camino con más retardo del circuito desde una entrada externa a través de puertas hasta una salida del circuito.

(b) Encuentre el camino con más retardo en el circuito desde una entrada externa hasta el flanco positivo del reloj.

(c) Encuentre el camino con más retardo desde el flanco positivo del reloj hasta la salida.

(d) Encuentre el camino con más retardo entre dos flancos de reloj positivos.

(e) Determine la frecuencia máxima de funcionamiento del circuito en megahercios (MHz).

6-12. Repita el Problema 6-11 suponiendo que el circuito está formado por dos circuitos idénticos al de la Figura 6-38 con la entrada *X* del segundo circuito conectada a la entrada *Y* del primer circuito.

6-13. En base al circuito secuencial de la Figura 6-17.

(a) Añada la lógica y/o las conexiones necesarias al circuito para proporcionar un reset asíncrono hacia el estado $A = 0$, $B = 1$ cuando la señal Reset = 1.

(b) Añada la lógica y/o las conexiones necesarias al circuito para proporcionar un reset síncrono hacia el estado $A = 0$, $B = 0$ cuando Reset = 0.

6-14. *Diseñe un circuito secuencial con dos flip-flop tipo *D* denominados *A* y *B* y una entrada *X*. Cuando $X = 0$, el estado del circuito permanece igual. Cuando $X = 1$, el circuito evoluciona a través de las siguientes transiciones de estado: desde 00 a 10 a 11 a 01, de nuevo a 00, y entonces se repiten.

6-15. *Se debe diseñar un complementador a 2 serie. Se presenta en la entrada *X* del complementador de dos bits un número entero binario de longitud arbitraria, el bit menos significativo se presenta en primer lugar. Cuando un bit determinado se presenta en la entrada *X*, el bit correspondiente aparece durante el mismo ciclo de reloj en la salida *Z*. Para indicar que una secuencia ha finalizado y que hay que inicializar el circuito para recibir otra secuencia, la entrada *Y* debe colocarse a 1 durante un ciclo de reloj. De lo contrario, Y es 0.

(a) Encuentre el diagrama de estados para el complementador a 2 serie.

(b) Encuentre la tabla de estados para el complementador serie.

6-16. El enlace USB (*Universal Serial Bus*) requiere un circuito que produzca la secuencia 00000001. Debe diseñar un circuito secuencial síncrono que comience produciendo esta secuencia para la entrada $E = 1$. Una vez que se comience la secuencia se debe comple-

tar. Si $E = 1$ durante la última salida de la secuencia, se repite la secuencia. Si no, si $E = 0$, la salida permanece constante a 1.

(a) Dibuje el Diagrama de estados de Moore para el circuito.
(b) Encuentre la tabla de estados y realice una asignación de estados.
(c) Diseñe el circuito usando flip-flop *D* y las puertas lógicas necesarias. Se debe incluir un Reset para colocar al circuito en el estado inicial apropiado en el cual se evalúe *E* para determinar si se debe producir la secuencia o una cadena constante de 1.

6-17. Repita el Problema 6-16 para la secuencia 01111110 que se utiliza en un determinado protocolo de comunicaciones de red.

6-18. + La secuencia del Problema 6-17 es un flag empleado en las comunicaciones de red y representa el principio de un mensaje. Este flag ha de ser único. Por consiguiente, deben aparecer menos de cinco 1 en cualquier otro lugar dentro del mensaje. Puesto que esto no es realista en mensajes normales, se recurre a un truco denominado inserción de ceros. El mensaje normal, que puede contener secuencias de más de cinco 1s, se introduce en la entrada *X* de un circuito secuencial de inserción de 0s. El circuito tiene dos salidas *Z* y *S*. Cuando aparece el quinto 1 en *X*, se inserta un 0 en la secuencia de salida que aparece en *Z* al tiempo que la salida $S = 1$ indica que el circuito de inserción de ceros está funcionando y que el circuito de entrada debe detenerse, no generando una nueva entrada durante un ciclo de reloj. Esto es necesario puesto que la inserción de 0 en la secuencia de salida la hace ser más larga que la secuencia de entrada sin la parada. Se ilustra la inserción de ceros mediante las siguientes secuencias de ejemplo:

Secuencia en *X* sin parada:	011111001111111000010111110101
Secuencia en *X* con las paradas:	0111110011111111000010111110101
Secuencia en *Z*:	0111110001111101100001011110101
Secuencia en *S*:	0000001000000010000000000000000

(a) Encuentre el diagrama de estados para el circuito.
(b) Encuentre la tabla de estados para el circuito y realice una asignación de estados.
(c) Encuentre una implementación del circuito usando flip-flop *D* y puertas lógicas.

6-19. En muchos de los sistemas de comunicación y de redes la señal transmitida por la línea de comunicación emplea un formato de no retorno a cero (NRZ). USB utiliza una versión específica denominada sin retorno a cero invertido (NRZI). Diseñe un circuito que convierta cualquier mensaje formado por una secuencia de 0 y 1 a una secuencia en formato NRZI. El mapeado para dicho circuito es el siguiente:

(a) Si el bit del mensaje es un 0, entonces el mensaje en formato NRZI cambia de inmediato de 1 a 0 o de 0 a 1, dependiendo del valor actual de NRZI.
(b) Si el bit del mensaje es un 1, entonces el mensaje en formato NRZI permanece fijo a 0 o a 1, dependiendo del valor actual de NRZI.

Estos cambios se muestran en el siguiente ejemplo suponiendo que el valor inicial del mensaje de NRZI es 1:

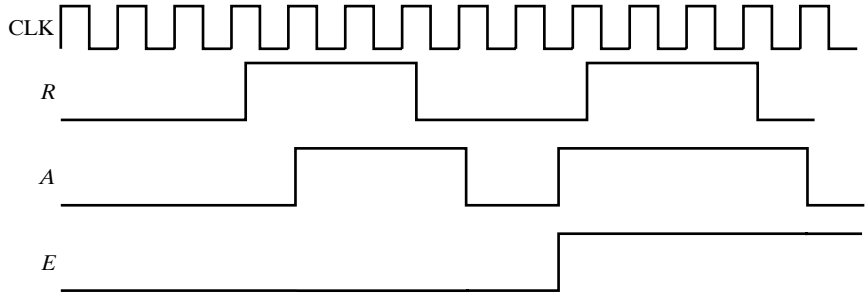
Mensaje:	10001110011010
Mensaje de NRZI:	10100001000101

(a) Encuentre el Diagrama de estados de Mealy para el circuito.
(b) Encuentre la tabla de estados para el circuito y realice una asignación de estados.
(c) Encuentre una implementación del circuito empleando flip-flop *D* y puertas lógicas.

6-20. +Repita el Problema 6-19, diseñando el circuito secuencial que transforma un mensaje de NRZI en un mensaje normal. El mapeado para dicho circuito es el siguiente:

(a) Si en el mensaje de NRZI aparece un cambio de 0 a 1 o de 1 a 0 entre bits adyacentes en el mensaje de NRZI, entonces el bit del mensaje es un 0.
(b) Si no se produce ningún cambio entre bits adyacentes del mensaje NRZI, entonces el bit del mensaje es un 1.

6-21. Se emplean las señales *Request* (*R*) y *Acknowledge* (*A*) para coordinar transacciones entre una CPU y su sistema de entrada-salida. Usualmente se denomina *handshake* al intercambio de estas señales. Son señales síncronas con el reloj y, para una transacción, sus transiciones aparecen siempre en el orden que se muestra en la Figura 6-39. Se debe diseñar un comprobador de *handshake* que verificará el orden de las transiciones. El comprobador tiene dos entradas *R* y *A*, una señal de reset asíncrona RESET y la salida



□ **FIGURA 6-39**
Cronograma para el Problema 6-21

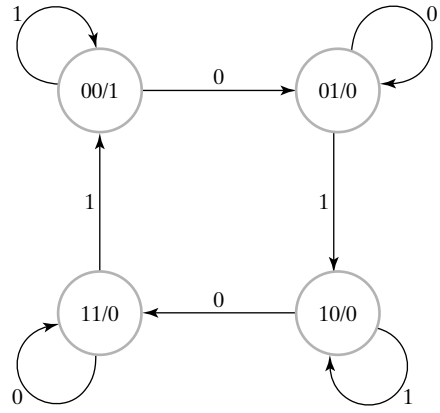
de Error (*E*). Si las transiciones están en orden, $E = 0$. Si las transiciones no están en orden, entonces *E* se convierte en 1 hasta que se aplique una señal asíncrona reset (RESET = 1) a la CPU.

(a) Encuentre el diagrama de estados para el comprobador de *handshake*.
(b) Encuentre la tabla de estados para el comprobador de *handshake*.

6-22. Debe diseñar un detector serie de 1. En la entrada *X* se aplica un número entero binario de longitud arbitraria empezando por el bit menos significativo. Cuando un determinado bit se presenta en la entrada *X*, aparece en la salida *Z*, durante el mismo ciclo de reloj, el bit correspondiente de salida. Mientras que los bits aplicados a *X* sean 0, $Z = 0$. Cuando se aplica a *X* el primer 1, $Z = 1$. Para todos los valores de bits aplicados a *X* después del primer 1, $Z = 0$. Para indicar que la secuencia ha finalizado y que el circuito debe ser inicializado para recibir otra secuencia, *Y* ha de ser 1 durante un ciclo de reloj. De lo contrario, *Y* es 0.

(a) Encuentre el diagrama de estados para el detector serie de 1.
(b) Encuentre la tabla de estados para el detector serie de 1.

6-23. *Un circuito secuencial tiene dos flip-flops *A* y *B*, una entrada *X* y una salida *Y*. El diagrama de estados se muestra en la Figura 6-40. Diseñe el circuito con flip-flops tipo *D*.



□ **FIGURA 6-40**
Diagrama de estados para el Problema 6-23

6-24. *Un flip-flop maestro-esclavo dominante a 1 tiene entradas de set y reset. Se diferencia de un flip-flop *SR* maestro-esclavo convencional en que, cuando *S* y *R* son iguales a 1, el flip-flop se pone a 1.

(a) Obtenga la tabla característica del flip-flop dominante a 1.
(b) Encuentre el diagrama de estados para el flip-flop dominante a 1.
(c) Diseñe el flip-flop empleando un flip-flop *SR* y puertas lógicas (incluyendo inversores).

6-25. Encuentre el diagrama lógico para el circuito que verifica la tabla de estados dada en la Tabla 6-5. Utilice flip-flops *D*.

6-26. +En la Tabla 6-11 se muestra la tabla de estados para un Contador Johnson. Este circuito no tiene ninguna entrada, y sus salidas son las salidas de los flip-flops. Puesto que no tiene ninguna entrada, siempre que se produzca un pulso de reloj, avanza de estado en estado.

(a) Diseñe el circuito empleando flip-flops *D* y suponga que los estados siguientes a los no especificados son indiferencias.
(b) Añada la lógica necesaria al circuito para inicializarlo al estado 000 en el reset inicial.
(c) En la subsección «Diseñando con estados no deseados» de la Sección 6-5, se discutieron tres técnicas para atender las situaciones en las que un circuito entra accidentalmente en un estado no usado. ¿Si el circuito que usted diseñó en las secciones (a) y (b) se utiliza en el juguete de un niño, cuál de las tres técnicas emplearía? Justifique su respuesta.
(d) De acuerdo con su respuesta en el Apartado (c), reajuste el circuito en caso de necesidad.
(e) Repita el Apartado (c) para el caso en el que el circuito se emplee para controlar los motores en un avión comercial de pasajeros. Justifique su respuesta.
(f) Repita el Apartado (d) en base a la respuesta del Apartado (e).

6-27. Realice una verificación manual de la solución (la suya o la que está el sitio web del libro) del Problema 6-24. Considere que todas las transiciones de *S* y *R* ocurren con el reloj igual a 0.

□ **TABLA 6-11**
Tabla de estados para el Problema 6-26

Estado actual	Estado futuro
ABC	***ABC***
000	100
100	100
110	111
111	011
011	001
001	000

6-28. Realice una verificación basada en la simulación lógica de su diseño para el Problema 6-25. La secuencia de entrada empleada en la simulación debe incluir todas las transiciones de la Tabla 6-6. La salida de la simulación incluirá la entrada *X*, las variables *A*, *B* y la salida *Z*.

6-29. *Genere una secuencia de verificación para el circuito descrito por la tabla de estados de la Tabla 6-10. Para reducir la longitud de la secuencia de la simulación, suponga que el simulador puede manejar entradas X y utilice X siempre que sea posible. Suponga que se dispone de una entrada reset para inicializar el estado a $A = 0$, $B = 0$ y que se deben ejecutar todas las transiciones del diagrama de estados.

6-30. Diseñe el circuito especificado por la Tabla 6-10 y utilice la secuencia del Problema 6-29 (la suya o la que se muestra en el sitio web del libro) para realizar una verificación basada en la simulación lógica de su diseño.

6-31. *Obtenga un cronograma similar al de la Figura 6-11 para un flip-flop *JK* disparado por flanco positivo durante cuatro impulsos de reloj. Muestre la evolución temporal de las señales *C*, *J*, *K*, *Y* y *Q*. Suponga que la salida *Q* es inicialmente igual a 1, con $J = 0$ y $K = 1$ para el primer pulso. Entonces, para los pulsos sucesivos, *J* pasa a 1, seguido por *K* que pasa a 0 y entonces *J* retorna de nuevo a 0. Suponga que cada entrada cambia cerca del flanco negativo del pulso.

Todos los archivos HDL para circuitos referidos en los restantes problemas están disponibles en ASCII para su simulación y edición en el sitio web del libro. Para los problemas que piden simulación se necesita un compilador/simulador de VHDL o Verilog. En cualquier caso, siempre se pueden escribir las descripciones HDL de muchos problemas sin necesidad de compilar o simular.

6-32. *Escriba una descripción VHDL para el multiplexor de la Figura 4-14 empleando un proceso que contenga una declaración **case** además de sentencias de asignación continuas tal y cómo se indicó en la Sección 4-7.

6-33. Repita el Problema 6-32 empleando un proceso VHDL con sentencias **if-then-else**.

6-34. + Escriba una descripción VHDL para el circuito secuencial que tiene el diagrama de estados dado en la Figura 6-25(d). Incluya una señal de RESET asíncrona para inicializar el circuito al estado `Init`. Compile su descripción, aplique una secuencia de entrada que pase a través de cada transición del diagrama de estados por lo menos una vez y

verifique los estados y la secuencia de salida comparándolos con los del diagrama de estados dado.

6-35. Escriba una descripción VHDL para el circuito especificado en el Problema 6-15.

6-36. Escriba una descripción VHDL para el circuito especificado en el Problema 6-19.

6-37. *Escriba una descripción VHDL para un flip-flop *JK* activo por flanco negativo con entrada de reloj CLK. Compile y simule su descripción. Aplique una secuencia que genere las ocho posibles combinaciones de las entradas *J* y *K* y del valor almacenado *Q*.

6-38. Escriba una descripción Verilog para el multiplexor de la Figura 4-14 empleando un proceso con una declaración **case** además de las sentencias de asignación continuas tal y cómo se ilustró en la Sección 4-8.

6-39. *Repita el Problema 6-38 utilizando un proceso Verilog que contenga sentencias **if-else**.

6-40. + Escriba una descripción Verilog para el circuito secuencial dado por el diagrama de estados de la Figura 6-25(d). Incluya una señal asíncrona de RESET para inicializar el circuito al estado `Init`. Compile su descripción, aplique una secuencia de entrada que pase a través de cada transición del diagrama de estados por lo menos una vez y verifique los estados y la secuencia de salida comparándolos con los del diagrama de estados dado.

6-41. Escriba una descripción Verilog para el circuito especificado en el Problema 6-15.

6-42. Escriba una descripción Verilog para el circuito especificado en el Problema 6-19.

6-43. *Escriba una descripción Verilog para un flip-flop *JK* activo por flanco negativo con entrada de reloj CLK. Compile y simule su descripción. Aplique una secuencia que genere las ocho posibles combinaciones de las entradas *J* y *K* y del valor almacenado *Q*.

CAPÍTULO

7

Registros y transferencia de registros

En los Capítulos 4 y 5 hemos estudiado los bloques de funciones combinacionales. En el Capítulo 6 hemos examinado los circuitos secuenciales. En este capítulo traemos ambas ideas juntas y presentamos los bloques de funciones secuenciales, generalmente conocidas como registros y contadores. Los circuitos que fueron analizados o diseñados en el Capítulo 6 no tenían ninguna estructura en particular y el número de flip-flops fue bastante pequeño. Por contra, los circuitos que aquí consideramos tienen más de una estructura, con múltiples etapas o células que son idénticas o casi idénticas. Además, debido a esta estructura, es fácil añadir más etapas para crear circuitos con muchos más flip-flops que en los circuitos descritos en el Capítulo 6. Los registros son particularmente útiles para almacenar información durante el procesado de datos y los contadores ayudan en la secuencia de estos procesamientos.

Un sistema digital presenta frecuentemente una ruta de datos y una unidad de control en el nivel más alto de la jerarquía de diseño. Una *ruta de datos* consiste en una lógica de procesamiento y una colección de registros que realizan el procesado de los datos. Una *unidad de control*, realizada con lógica, determina la secuencia del procesado de datos realizada por la ruta de datos. La notación Transferencia de Registros describe las acciones elementales para el procesado de datos, llamada *microoperaciones*. La transferencia de registros es el movimiento de información entre registros, entre registros y memoria, a través de la lógica de procesamiento. El hardware dedicado a transferir datos usando multiplexores y el hardware compartido para esta transferencia, llamado bus, realizan el movimiento de los datos.

En la computadora genérica del Capítulo 1, los registros se usan de forma extensiva para almacenamiento temporal de datos en zonas aparte de la memoria. Este tipo de registros son, frecuentemente, grandes, con al menos 32 bits. Hay registros especiales, llamados registros de desplazamiento, que se usan con menos frecuencia, que aparecen principalmente en los bloques de entrada/salida del sistema. Los contadores se usan en varias partes de la computadora para controlar o seguir la pista a las secuencias de las operaciones. En general, los bloques funcionales secuenciales se usan ampliamente en la computadora genérica. En particular, tanto la CPU, como la FPU del procesador contienen gran número de registros que están involucrados en la transferencia de registros y la ejecución de las microoperaciones. En la CPU y la FPU es donde tiene lugar la transferencia de datos, sumas, restas y otras microoperaciones. Finalmente, las conexiones mostradas entre las diversas partes electrónicas de la computadora son los buses, se discuten por primera vez en este capítulo.

7-1 REGISTROS Y HABILITACIÓN DE CARGA

Un registro está compuesto por un conjunto de flip-flops. Puesto que cada flip-flop es capaz de almacenar un bit de información, un registro de *n* bits, compuesto de *n* flip-flops, es capaz de almacenar *n* bits de información binaria. Para una definición más general, un *registro* está compuesto de un conjunto de flip-flops, junto con puertas que definen la transición de sus estados. Esta definición también incluye a los diversos circuitos secuenciales considerados en el Capítulo 6. Más comúnmente, el termino *registro* se aplica a un conjunto de flip-flops, con la posibilidad de incorporar puertas, que realicen las tareas de procesamiento de datos. Los flip-flops retienen los datos y las puertas determinan el dato nuevo, o transformado, que se transfiere a los flip-flops.

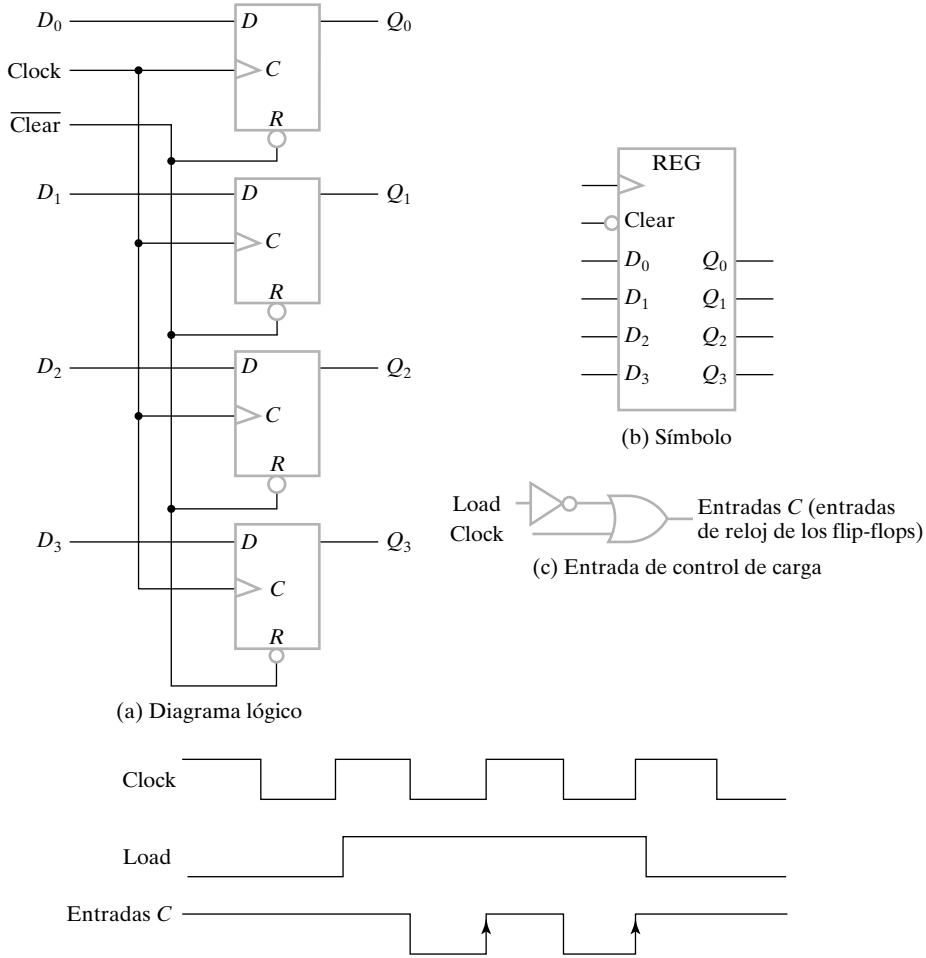
Un contador es un registro que transita a través de una determinada secuencia de estados según la aplicación de pulsos de reloj. Las puertas en un contador se conectan de tal forma que prescribe la secuencia de los estados binarios. Aunque los contadores son un tipo especial de registros, habitualmente se diferencian de los registros.

Los registros y los contadores son bloques funcionales secuenciales que se usan extensivamente en el diseño de sistemas digitales en general y en el diseño de computadoras en particular. Los registros son útiles para almacenamiento y manipulación de la información; los contadores se emplean en los circuitos que secuencian y controlan las operaciones en un sistema digital.

El registro más simple es un registro compuesto solamente por flip-flops sin puertas adicionales. La Figura 7-1(a) muestra un registro construido con cuatro flip-flops de tipo *D*. El reloj, común a los flip-flops, los dispara con el flanco de subida de cada pulso, y la información binaria disponible en las entradas *D* se transfiere a dentro del registro de 4 bits. Las cuatro salidas, *Q*, pueden ser muestreadas para obtener la información binaria almacenada en el registro. La entrada $\overline{\text{Clear}}$ va conectada a la entrada $\bar{R}$ de los cuatro flip-flops y se usa para inicializar el registro con valor 0 antes de operar con el reloj. Esta entrada, se etiqueta $\overline{\text{Clear}}$ en lugar de *Clear*, puesto que ha de aplicarse un 0 al reset de los flip-flops asíncronamente. La activación de las entradas de reset asíncrono, $\bar{R}$, durante el modo síncrono de operación (con el reloj) puede llevar a diseño de circuitos que son muy dependientes de retardos y pueden, por tanto, funcionar mal con facilidad. Así, mantendremos el $\overline{\text{Clear}}$ a un nivel lógico 1 durante el modo normal de operación sincronizado, permitiendo que sea un 0 lógico cuando se desea una inicialización del sistema. Véase que la posibilidad de poner el registro a 0 es opcional; el que pueda ponerse un registro a 0 va a depender de si el registro del sistema tiene entrada de inicialización o no.

La transferencia de nueva información en el registro se denomina *carga* del registro. Si todos los bits del registro se cargan simultáneamente con el pulso de reloj común, decimos que la carga se realiza en paralelo. Un flanco positivo o de subida aplicado a la entrada *Clock* de (reloj) del registro de la Figura 7-1(a) carga todas las entradas *D* de los flip-flops en paralelo.

La Figura 7-1(b) muestra el símbolo que representa el registro de la Figura 7-1(a). Este símbolo permite el uso de un registro en un diseño jerárquico. El símbolo tiene todas las entradas del circuito a su izquierda y todas las salidas del circuito a la derecha. Las entradas incluyen la entrada de reloj con el indicador que representa el disparo con el flanco positivo de los flip-flops. Véase que el nombre Clear aparece dentro del símbolo con una burbuja en la línea de la señal en el exterior del símbolo. Esta notación indica que la aplicación de un 0 lógico a esta señal activa la operación de puesta a 0 de los flip-flops del registro. Si la línea de la señal fuese etiquetada fuera del símbolo, debería aparecer como $\overline{\text{Clear}}$.



□ **FIGURA 7-1**
Registro de 4 bits

Registro con carga en paralelo

La mayoría de los sistemas digitales tienen un generador de reloj maestro que proporciona un tren de pulsos continuo. Los pulsos se aplican a todos los flip-flops y registros del sistema. De hecho, el reloj maestro actúa como el corazón que proporciona un pulso constante a todas las parte del sistema. En el diseño de la Figura 7-1(a), se debe impedir que el reloj actúe sobre la entrada de reloj del circuito si se quiere dejar inalterado el contenido del registro. Para ello, se usa una señal de control aparte para controlar que los ciclos de reloj afecten al registro. Así se evita que los pulsos de reloj lleguen al registro si su contenido no ha de ser cambiado. Esta función puede realizarse con una entrada de control de carga, Load, combinada con el reloj, como se muestra en la Figura 7-1(c). La salida de la puerta OR se aplica a las entradas *C* de los flip-flops del registro. La ecuación de la lógica mostrada es

$$C \text{ entradas} = \overline{Load} + Clock$$

Si la señal Load es 1, $C = Clock$, así el registro se sincroniza normalmente y la nueva información se transfiere al registro con el flanco de subida del reloj. Si la señal Load es 0, $C = 1$. Con esta entrada constante aplicada no hay flancos positivos en las entradas *C*, así el contenido del registro permanece sin cambiar. El efecto de la señal Load sobre la señal *C* de los flip-flops se muestra en la Figura 7-1(d). Véase que los pulsos de reloj que aparecen en *C* son pulsos que transitan a 0 seguidos de un flanco positivo que dispara a los flip-flops. Estos pulsos y flancos aparecen cuando Load es 1 y cuando Load es 0 se reemplazan por un 1 constante. Para que este circuito funcione correctamente, Load debe ser constante en su valor seleccionado, tanto 0 como 1, durante el intervalo en que Clock es 0.

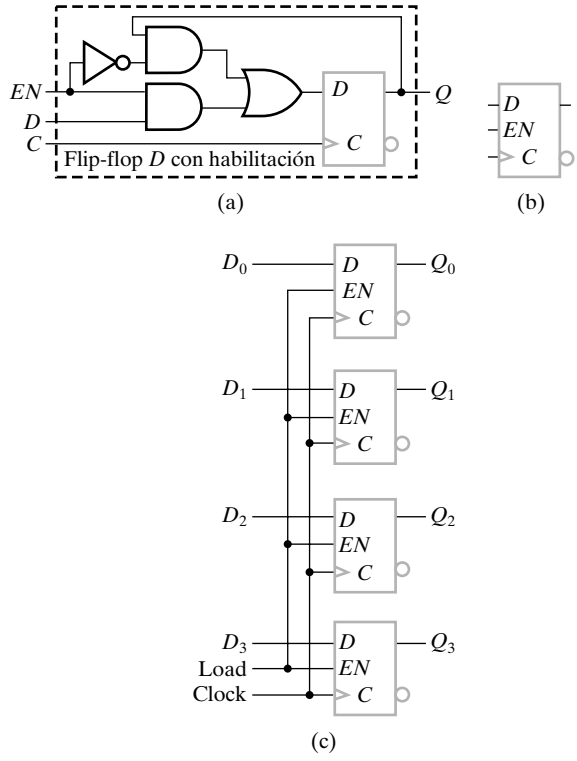
Esta situación ocurre cuando Load procede de un flip-flop que se dispara en el flanco positivo del reloj, circunstancia normal si todos los flip-flops del sistema son activados con el flanco positivo. A esta técnica que deja pasar o no la señal de reloj a las entradas *C* del registro usando puertas lógicas, se le llama *«clock gating»*[1].

La inserción de puertas en el camino del reloj produce diferentes retardos de propagación entre el Reloj y las entradas de los flip-flops con o sin *clock gating*. Si la señal de reloj llega a diferentes flip-flops o registros en diferentes instantes de tiempo, se dice que se produce un *skew* de reloj. Pero para tener un sistema síncrono fiable, debemos asegurar que todos los pulsos de reloj llegan simultáneamente a todo el sistema de forma que todos los flip-flops se disparan al mismo tiempo. Por esta razón, en diseños rutinarios, se aconseja el control de la operación del registro sin pasar el reloj por puertas. Por otra parte, los retardos deben controlarse para conseguir un *skew* de reloj tan cercano como sea posible. Este concepto es aplicable en diseños de bajo consumo y de alta velocidad.

En la Figura 7-2(c), se muestra un registro de 4 bits con entrada de control de carga llevada a cabo mediante puertas y sobre las entradas *D* de los flip-flops, en lugar de hacerlo sobre las entradas *C*. Este registro se basa en una célula de un bit, mostrada en la Figura 7-2(a), que consiste en un multiplexor de 2 a 1 y un flip-flop tipo *D*. La señal *EN* selecciona entre la entrada del bit *D* de la célula y el valor de *Q* de la salida de la célula. Para $EN = 0$, se selecciona *Q* y la salida se recarga en el flip-flop, preservando su estado actual. La realimentación de la salida a la entrada es necesaria puesto que el flip-flop tipo *D*, no como otros tipos de flip-flops, no tiene una condición de entrada de «no cambio». Con cada pulso de reloj, la entrada *D* determina el siguiente estado de la salida. Para dejar la salida sin modificar es necesario hacer la entrada *D* igual al valor presente en la salida. La lógica de la Figura 7-2(a) puede verse como un nuevo flip-flop tipo *D*, un flip-flop tipo *D* con habilitación, cuyo símbolo se muestra en la Figura 7-2(b).

El registro se diseña colocando cuatro flip-flops con habilitación en paralelo y conectando la entrada de carga, Load, a la entrada *EN*. Cuando Load es 1, el dato colocado en las cuatro entradas se transfiere al registro con el siguiente flanco de subida del reloj. Si Load es 0, el valor actual del registro permanece en el siguiente flanco de subida del reloj. Véase que los pulsos de reloj se aplican a las entradas *C* continuamente. La señal Load determina si el siguiente pulso acepta la nueva información o deja la información del registro intacta. La transferencia de información de las entradas a registrar se hace simultáneamente para los cuatro bits durante una sola transición positiva del pulso. Este método de transferencia es preferible, tradicionalmente, frente al *clock gating*, puesto que evita el *skew* de reloj y los posibles fallos de funcionamiento del circuito.

[1] *N. del T.*: No existe un término en español para denominar a esta técnica.

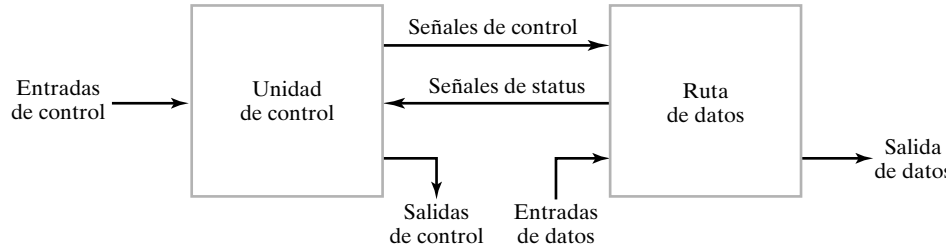


□ **FIGURA 7-2**
Registro de 4 bits con carga en paralelo

7-2 TRANSFERENCIA DE REGISTROS

Un sistema digital es un circuito secuencial construido con flip-flops y puertas interconectados. En el Capítulo 6 aprendimos que los circuitos secuenciales pueden especificarse por medio de tablas de estados. Especificar un sistema digital grande con tablas de estados es muy difícil, sino imposible, debido al prohibitivo gran número de estados. Para vencer esta dificultad, los sistemas digitales se diseñan usando esquemas modulares y jerárquicos. El sistema se divide en subsistemas o módulos, cada uno de los cuales desarrolla alguna función. Los módulos se construyen jerárquicamente a partir de bloques funcionales como registros, contadores, decodificadores, multiplexores, buses, elementos aritméticos, flip-flops y puertas. Los diversos subsistemas se comunican con señales de datos y de control para formar los sistemas digitales.

En la mayoría de los diseños de sistemas digitales, partimos el sistema en dos tipos de módulos: una *ruta de datos*, que realiza las operaciones de procesado de datos, y una *unidad de control*, que determina la secuencia de estas operaciones. La Figura 7-3 muestra, en general, la relación entre una ruta de datos y una unidad de control. Las señales de control son señales binarias que activan las diversas operaciones para el procesamiento de datos. Para activar la secuencia de tales operaciones, la unidad de control manda la secuencia adecuada de señales de control a la ruta de datos. La unidad de control, en cambio, recibe los bits de status que describen aspectos del estado de la ruta de datos. La unidad de control usa los bits de status para definir la secuencia específica de operaciones que se van a llevar a cabo. Observe que la ruta de datos y la unidad de control pueden también interactuar con otras partes de un sistema digital,



□ **FIGURA 7-3**
Interacción entre la ruta de datos y la unidad de control

como la memoria, lógica de entrada/salida, mediante los datos de entrada, datos de salida, entradas y salidas de control.

Las rutas de datos se definen mediante sus registros y las operaciones realizadas sobre datos binarios almacenados en los registros. Ejemplos de estas operaciones en registros son: carga, inicialización, desplazamiento y cuenta. Se considera que los registros son los componentes básicos de un sistema digital. Se llama *operaciones de transferencia de registros* al movimiento de los datos almacenados en los registros y al procesado realizado sobre los datos. Las operaciones de transferencia de registros de un sistema digital se especifican mediante tres componentes básicos siguientes:

1. conjunto de registros del sistema,
2. operaciones que se realizan sobre los datos almacenados en los registros, y
3. el control que supervisa la secuencia de operaciones del sistema.

Un registro tiene la capacidad de realizan una o más *operaciones elementales* como carga, cuenta, suma, resta y desplazamiento. Por ejemplo, un registro de desplazamiento a la derecha es un registro que puede desplazar un dato a la derecha. Un contador es un registro que incrementa un número en uno. Un flip-flop es un registro de 1 bit que puede ser puesto a uno o a cero síncronamente con una señal de reloj. De hecho, según esta definición, a los flip-flops y puertas asociadas a él, en cualquier circuito secuencial, se le llaman registros.

Una operación elemental desarrollada sobre los datos almacenados en los registros se le denomina *microoperación*. Ejemplos de microoperaciones son: la carga del contenido de un registro en otro, suma del contenido de dos registros e incremento del contenido de un registro. Una microoperación normalmente se desarrolla, aunque no siempre, en paralelo sobre un vector de bits durante un ciclo de reloj. El resultado de la microoperación puede reemplazar los datos anteriores que había en el registro. Opcionalmente, el resultado puede ser transferido a otro registro, dejando los datos previos inalterados. Los bloques funcionales secuenciales presentados en este capítulo son registros que realizan una o más microoperaciones.

La unidad de control proporciona las señales que realizan la secuencia de microoperaciones de una forma determinada. El resultado de la operación en curso puede determinar tanto la secuencia de señales de control, como la secuencia de futuras microoperaciones a ser ejecutada. Véase que el término «microoperación, tal como se utiliza aquí, no se refiere a ninguna forma en particular de generar las señales de control: exactamente hablando, no implica que las señales de control se generan a partir de una unidad de control basada en una técnica llamada microprogramación.

Este capítulo presenta los registros, su realización y la transferencia de registros usando un sencillo lenguaje de transferencia de registros (RTL) para representar y especificar las operaciones sobre sus contenidos. El lenguaje de transferencia de registros utiliza un conjunto de expre-

siones y sentencias que se parecen a las sentencias usadas en los HDLs y en los lenguajes de programación. Esta notación puede especificar concisamente parte o todo un sistema digital complejo como es un procesador. La especificación sirve pues como base para un diseño más detallado de un sistema.

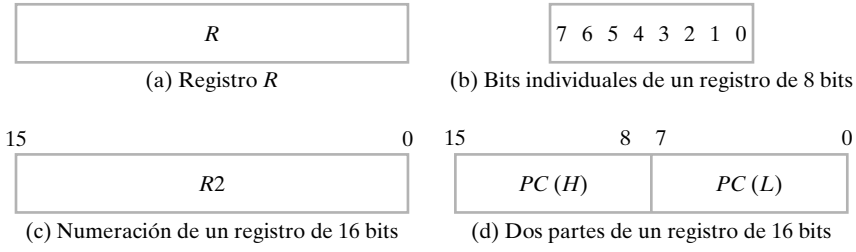
7-3 OPERACIONES DE TRANSFERENCIA DE REGISTROS

Designaremos a los registros de un sistema digital mediante letras en mayúsculas (seguidas a veces por números) que indican la función del registro, por ejemplo, un registro que contiene una dirección de una memoria se llama, normalmente, como registro de direcciones y se le puede designar como *AR* (del inglés *Address Register*). Otras formas de llamar a un registro son: *PC* para el contador de programa (del inglés *Program Counter*), *IR* para el registro de instrucciones (del inglés *Instruction Register*), y *R*2 para el registro número 2. Los flip-flops de un registro de *n* bits se nombran típicamente, de forma individual, mediante la secuencia de 0 hasta *n*-1, empezando por el 0 para la posición del bit menos significativo (normalmente el más a la derecha) e incrementándose hacía el de la posición más significativa. Si el bit 0 es el de la derecha, a esta ordenación se le llama *little-endian*, de la misma forma como para los bytes del Capítulo 1. El orden inverso, con el bit 0 a la izquierda, se le denomina como *big-endian*. La Figura 7-4 muestra las representaciones de los registros en forma de diagrama de bloques. La forma más común de representar un registro es una caja rectangular con el nombre del registro dentro, como en la parte (a) de la figura. A cada uno de los bits se le puede identificar como en la parte (b) de la figura. Otra forma es representando sólo los valores de los bits más a la izquierda y más a la derecha encima de la caja del registro, como se ilustra en el registro de 16 bits, *R*2, en la parte (c). Un contador de programa de 16 bits, *PC*, se puede dividir en dos secciones como en la parte (d) de la figura. En este caso, con el símbolo *L* (del inglés *low-order byte*) para designar los bits de 0 a 7, y el símbolo *H* (del inglés *high-order byte*) para los bits de 8 hasta 15. La etiqueta *PC*(*L*), que también puede escribirse como *PC*(7:0), refiriéndose al byte de menor peso, y *PC*(*H*) o *PC*(15:8) para hacer referencia al byte de mayor peso.

La forma simbólica de representar la transferencia de un dato de un registro a otro se hace mediante la colocación del operador (←). Así la sentencia

$$R2 \leftarrow R1$$

indica la transferencia del contenido del registro *R*1 al registro *R*2. En otras palabras, la sentencia indica la copia del contenido de *R*1 en *R*2. El registro *R*1 es la *fuente* de la transferencia y el registro *R*2 es el *destino*. Por definición, el contenido del registro fuente no cambia como resultado de la transferencia; sólo el contenido del registro destino, *R*2, cambia.



(a) Registro *R*
(b) Bits individuales de un registro de 8 bits
(c) Numeración de un registro de 16 bits
(d) Dos partes de un registro de 16 bits

□ **FIGURA 7-4**
Diagrama de bloques de registros

Una sentencia que especifica una transferencia de un registro implica que el circuito de la ruta de datos esta disponible desde las salidas del registro fuente hasta las entradas del registro destino y que éste tiene la capacidad de realizar una carga paralela. Normalmente, queremos que una transferencia dada ocurra no para cada pulso de reloj, sino para unos valores específicos de las señales de control. Esto puede definirse como una *sentencia condicional*, simbolizada con la expresión *if-then*.

$$if\ (K_1 = 1)\ then\ (R2 \leftarrow R1)$$

donde K_1 es una señal de control generada por la unidad de control. De hecho, K_1 puede ser cualquier función booleana que se evalúa a 0 o a 1. Una forma más concisa de escribir la expresión *if-then* es

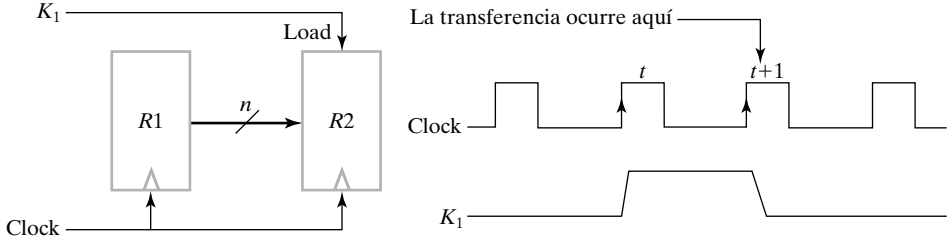
$$K_1\colon R2 \leftarrow R1$$

Esta condición de control, terminada con dos puntos, simboliza el requisito de que la operación de transferencia se ejecuta por el hardware sólo si $K_1 = 1$.

Cada sentencia escrita en notación de transferencia de registros presupone una construcción hardware para realizar dicha transferencia. La Figura 7-5 muestra un diagrama de bloques que describe la transferencia del *R*1 a *R*2. Las *n* salidas del registro *R*1 están conectadas a las *n* entradas del registro *R*2. La letra *n* se utiliza para indicar el número de bits que hay en el camino de transferencia de *R*1 a *R*2. Cuando la anchura del camino es conocido, *n* se sustituye por ese número. El registro *R*2 tiene una señal de control de entrada que se activa con la señal K_1. Se supone que la señal está sincronizada con el mismo reloj que el aplicado al registro. Se supone también que los flip-flops se disparan con el flanco positivo de este reloj. Como se muestra en el diagrama de tiempos, K_1 se pone a 1 con el flanco de subida del pulso de reloj en el instante *t*. La siguiente transición positiva del reloj en el instante $t + 1$ encuentra que $K_1 = 1$, y las entradas de *R*2 se cargan en el registro de forma paralela. En este caso, K_1 vuelve a 0 en el flanco positivo en el instante $t + 1$, de forma que se produce una única transferencia de *R*1 a *R*2.

Véase que el reloj no está incluido como variable en la sentencia de transferencia de registro. Se supone que todas las transferencias ocurren sincronizadamente como respuesta a una transición del reloj. Incluso cuando la condición K_1 se activa en el instante *t*, la transferencia no ocurre hasta que el registro se dispara por la siguiente transición positiva del reloj en el instante $t + 1$.

Los símbolos básicos que usamos en la notación de transferencia de registros se muestran en la Tabla 7-1. Los registros se designan mediante letras mayúsculas, seguidas posiblemente por una o más letras mayúsculas y números. Los paréntesis se usan para nombrar parte de un regis-



□ **FIGURA 7-5**
Transferencia de *R*1 a *R*2 cuando $K_1 = 1$

□ **TABLA 7-1**
Símbolos básicos para transferencia de registros

Símbolo	Descripción	Ejemplos
Letras (y números)	Indica un registro	*AR*, *R*2, *DR*, *IR*
Paréntesis	Indica parte de un registro	*R*2(1), *R*2(7:0), *AR*(L)
Flecha	Indica transferencia del dato	$R1 \leftarrow R2$
Coma	Separa transferencias simultáneas	$R1 \leftarrow R2$, $R2 \leftarrow R1$
Corchetes	Especifica una dirección de memoria	$DR \leftarrow M[AR]$

tro, especificando el rango de bits del registro o dando un nombre simbólico a una porción del registro. La flecha apuntando a la izquierda indica una transferencia de datos y la dirección de la transferencia. Se emplea una coma para separar dos o más transferencias que se ejecutan al mismo tiempo. Por ejemplo la sentencia

$$K_3\text{: } R2 \leftarrow R1,\ R1 \leftarrow R2$$

indica una operación que cambia el contenido de dos registros simultáneamente para un flanco positivo de reloj en el que $K_3 = 1$. Tal cambio es posible con registros hechos con flip-flops pero presenta dificultades por problemas de temporización si los registros están hechos con *latches*. Los corchetes se usan juntamente con una transferencia a memoria. La letra *M* indica una palabra de memoria y el registro encerrado en los corchetes proporciona la dirección de la palabra de la memoria. Esto se explica con más detalle en el Capítulo 10.

7-4 NOTA PARA USUARIOS DE VHDL Y VERILOG

Existen algunas similitudes con el lenguaje de transferencia de registros aquí presentado, tanto para VHDL como para Verilog. En particular, hay diferentes notaciones para cada uno de los tres lenguajes. La Tabla 7-2 compara la notación entre operaciones de transferencias de registros idénticas o similares de los tres lenguajes. Como este tema se estudia en este capítulo y en posteriores, esta tabla le ayudará a relacionar las descripciones en RTL y las correspondientes a VHDL o Verilog.

7-5 MICROOPERACIONES

Una microoperación es una operación básica realizada sobre los datos almacenados en registros o en memoria. Las microoperaciones encontradas más frecuentemente en sistemas digitales son de 4 tipos:

1. Microoperaciones de transferencia, transfiere datos en binomio de un registro a otro.
2. Microoperaciones aritméticas, realizan operaciones aritméticas en los datos de los registros.
3. Microoperaciones lógicas, realizan manipulación de los bits de los datos de los registros.
4. Microoperaciones de desplazamiento, desplazan los datos de los registros.

□ **TABLA 7-2**
Símbolos RTL, VHDL y Verilog para transferencias de registros

Operación	RTL	VHDL	Verilog
Asignación combinacional	=	< = (concurrente)	Assign = (nonblocking)
Transferencia de registro	←	< = (concurrente)	< = (nonblocking)
Suma +	+	+	+
Resta −	−	−	−
Bitwise AND	∧	and	&
Bitwise OR	∨	or	|
Bitwise XOR	⊕	xor	∧
Bitwise NOT	¯	not	∼
Desplazamiento a la izquierda (lógico)	sl	sll	≪
Desplazamiento a la derecha (lógico)	sr	srl	≫
Vectores/registros	A(3:0)	A(3 downto 0)	A[3:0]
Concatenación	‖	&	{ , }

N. del T.: Bitwise es una operación entre dos vectores bit a bit

Una microoperación dada puede ser de más de un tipo. Por ejemplo, la operación de complemento a 1 es una microoperación tanto aritmética como lógica.

Las microoperaciones de transferencia se presentaron en las secciones anteriores. Este tipo de microoperación no cambia los bits de datos cuando los mueven de un registro fuente a uno destino. Los otres tres tipos de operaciones pueden producir un dato nuevo en binomio, es decir, nueva información. En los sistemas digitales, los conjuntos de operaciones básicas se utilizan para formar secuencias que realizan operaciones más complicadas. En esta sección, definimos un conjunto básico de microoperaciones, notación simbólica para éstas, y las descripciones del hardware digital que las lleva a cabo.

Microoperaciones aritméticas

En esta sección definimos las operaciones aritméticas básicas tales como la suma, resta, incremento, decremento y el complemento. La sentencia

$$R0 \leftarrow R1 + R2$$

especifica una operación de suma. Expresa que el contenido del registro $R2$ se suma al contenido del registro $R1$ y la suma se transfiere al registro $R0$. Para realizar esta sentencia con hardware necesitamos tres registros y un componente combinacional que haga la suma, un sumador paralelo. Las restantes operaciones aritméticas se recogen en la Tabla 7-3. La resta se realiza frecuentemente mediante la complementación y la suma. En lugar de utilizar el operador menos, especificamos la resta mediante el complemento a 2 según la siguiente expresión

$$R0 \leftarrow R1 + \overline{R2} + 1$$

□ **TABLA 7-3**
Microoperaciones aritméticas

Designación simbólica	Descripción
$R0 \leftarrow R1 + R2$	El contenido de $R1$ más el contenido de $R2$ se transfiere a $R0$
$R2 \leftarrow \overline{R2}$	Complemento del contenido de $R2$ (complemento a 1)
$R2 \leftarrow \overline{R2} + 1$	Complemento a 2 del contenido de $R2$
$R0 \leftarrow R1 + \overline{R2} + 1$	$R1$ más el complemento a 2 de $R2$ se transfiere a $R0$ (resta)
$R1 \leftarrow R1 + 1$	Incrementa el contenido de $R1$ (cuenta ascendente)
$R1 \leftarrow R1 - 1$	Decrementa el contenido de $R1$ (cuenta descendente)

donde $\overline{R2}$ es el complemento a 1 de $R2$. Sumando 1 a $\overline{R2}$ conseguimos el complemento a 2 de $R2$. Finalmente, sumando el complemento a 2 de $R2$ al contenido de $R1$ obtenemos el equivalente a $R1 - R2$.

Las microoperaciones de incremento y decremento se simbolizan con operaciones de más uno y menos uno, respectivamente. Estas operaciones se llevan a cabo usando un circuito combinacional especial, un sumador-restador, o un contador binario ascendente-descendente con carga en paralelo.

La multiplicación y la división no están en la Tabla 7-3. La multiplicación se puede representar mediante el símbolo * y la división mediante /. Estas dos operaciones no están incluidas en el conjunto básico de microoperaciones aritméticas puesto que se asume que se realizan con secuencias de microoperaciones básicas. Por contra, la multiplicación puede considerarse como una microoperación si se realiza con un circuito combinacional como el que se ilustró en la Sección 5-4. En tal caso, el resultado se transfiere en el registro destino con el flanco de reloj después de que todas las señales se han propagado a través de todo el circuito combinacional.

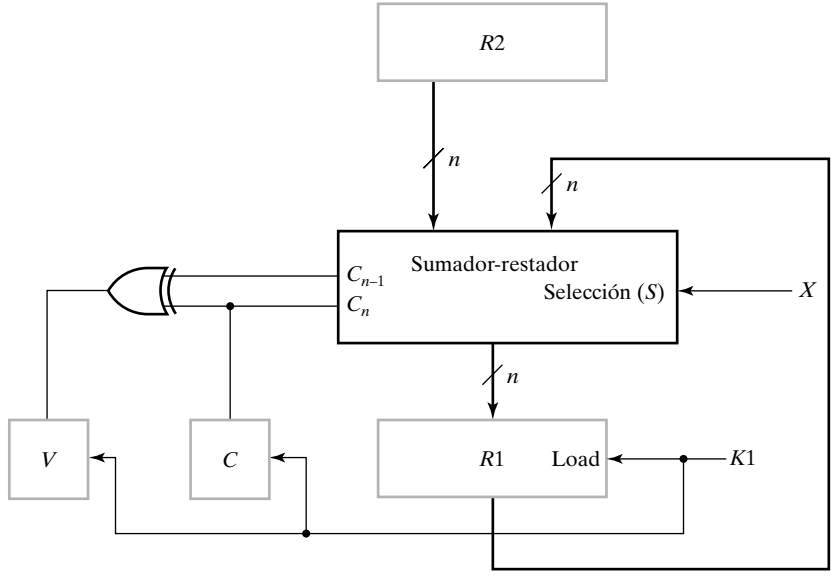
$$\bar{X}K_1 : R1 \leftarrow R1 + R2$$

$$XK_1 : R1 \leftarrow R1 + \overline{R2} + 1$$

La variable de control K_1 activa una operación de suma o resta. Si en el mismo instante de tiempo, la variable de control X es 0, entonces $\bar{X}K_1 = 1$, y el contenido de $R2$ se suma al contenido de $R1$. Si X es 1, entonces $XK_1 = 1$ y el contenido de $R2$ se resta del contenido de $R1$. Véase que las 2 condiciones de control son funciones booleanas, cuyo valor es 0 cuando $K_1 = 0$, condición que evita la ejecución de ambas operaciones simultáneamente.

En el diagrama de bloques de la Figura 7-6 se muestra la realización de las 2 expresiones anteriores. Un sumador-restador de n bits, similar al mostrado en la Figura 5-8, recibe por sus entradas los datos de los registros $R1$ y $R2$. La suma o resta se aplica a las entradas de $R1$. La entrada de selección S del sumador-restador elige la operación del circuito. Cuando $S = 0$, las dos entradas se suman, y cuando $S = 1$, $R2$ se resta de $R1$. Aplicando la variable de control X a la entrada S, se activa la operación solicitada. La salida del sumador-restador se carga en $R1$ con cualquier flanco de subida del reloj si $\bar{X}K_1 = 1$ o $XK_1 = 1$. Esto se pueda simplificar a K_1, puesto que

$$\bar{X}K_1 + XK_1 = (\bar{X} + X)K_1 = K_1$$



□ **FIGURA 7-6**
Realización de las microoperaciones de suma y resta

Así, la variable de control X selecciona la operación y la variable de control K_1 carga el resultado en $R1$.

Basándonos en la discusión sobre el *overflow* de la Sección 5-3, la salida de *overflow* se transfiere al flip-flop V, y la salida de acarreo de bit más significativo del sumador-restador se transfiere al flip-flop C, como se muestra en la Figura 7-6. Estas transferencias, de C y V, ocurren cuando $K_1 = 1$ y no se han representado en las sentencias de transferencia de registros; si se desea, podríamos indicarlas como transferencias adicionales simultáneas.

Microoperaciones lógicas

Las microoperaciones lógicas son útiles para manipular los bits almacenados en un registro. Estas operaciones consideran cada bit de un registro separadamente y los trata como valores binarios. Los símbolos para cada una de las cuatro operaciones lógicas básicas se muestran en la Tabla 7-4. La microoperación NOT, representada por una barra sobre el nombre del registro fuente, complementa todos los bits y es lo mismo que el complemento a 1. El símbolo $\wedge$ se usa para designar la microoperación AND y el símbolo $\vee$ designa la microoperación OR. Usando estos símbolos especiales, es posible distinguir entre la microoperación suma, representada por un $+$ y la microoperación OR. Aunque el símbolo $+$ tiene dos significados, se puede distinguir viendo donde aparecen. Si el $+$ aparece en una microoperacion, significa una suma. Si el $+$ aparece en una función booleana o de control, significa OR. Para la microoperación OR se usará siempre el símbolo $\vee$. Por ejemplo en la expresión

$$(K_1 + K_2)\colon R1 \leftarrow R2 + R3,\ R4 \leftarrow R5 \vee R6$$

el $+$ entre K_1 y K_2 es una operación OR entre dos variables en una condición de control. El $+$ entre $R2$ y $R3$ especifica una microoperación de suma. La microoperación OR se indica con el

□ **TABLA 7-4**
Microoperaciones lógicas

Designación simbólica	Descripción
$R0 \leftarrow \overline{R1}$	Bitwise lógico NOT (complemento a 1)
$R0 \leftarrow R1 \wedge R2$	Bitwise lógico AND (pone a 0 los bits)
$R0 \leftarrow R1 \vee R2$	Bitwise lógico OR (pone a 1 los bits)
$R0 \leftarrow R1 \oplus R2$	Bitwise lógico XOR (complementa bits)

símbolo $\vee$ entre los registros $R5$ y $R6$. Las microoperaciones lógicas pueden realizarse fácilmente con un conjunto de puertas, uno por cada bit. La NOT de un registro de n bits se obtiene con n puertas NOT en paralelo. La microoperación AND se obtiene usando un grupo de n puestas AND, cada una de las cuales recibe un par de entradas correspondiente a dos registros fuente. Las salidas de las puertas AND se aplican a las entradas correspondientes del registro destino. Las microoperaciones OR y la OR exclusiva necesitan una colocación similar.

Las microoperaciones pueden cambiar los valores de los bits, poner a cero un grupo de bits, o insertar un nuevo valor a un bit de un registro. Los siguientes ejemplos muestran cómo los bits almacenados en un registro, $R1$, de 16 bits pueden ser cambiados selectivamente usando una microoperación y operaciones lógicas, y ser almacenados en un registro, $R2$, de 16 bits.

La microoperación AND se puede usar para poner a 0 uno a más bits de un registro. Las ecuaciones booleanas $X \cdot 0 = 0$ y $X \cdot 1 = 1$ obligan a que, cuando se hace una operación AND con 0, una variable X produce un 0 pero cuando se hace una operación AND con 1, la variable permanece sin cambiar. Un bit o grupo de bits dados de un registro pueden ponerse a 0 si se hace una operación AND con 0. Considere el siguiente ejemplo:

10101101 10101011	$R1$	(dato)
00000000 11111111	$R2$	(máscara)
00000000 10101011	$R1 \leftarrow R1 \wedge R2$	

El operando de 16 bits de $R2$ tiene ceros en el byte de mayor peso y unos en el de menor peso. Realizando la operación AND del contenido de $R2$ con el de $R1$, es posible poner a cero el byte de mayor peso de $R1$ y dejar los bits del byte de menor peso inalterados. Así, la operación AND se puede utilizar para poner a cero los bits de un registro de forma selectiva. A esta operación se le suele llamar *enmascaramiento* de bits, puesto que enmascara o borra todo los unos en el dato de $R1$ según la posición de los bits que sean 0 en la máscara proporcionada por $R2$.

La microoperación OR se usa para poner a 1 uno o más bits de un registro. Las ecuaciones booleanas $X + 1 = 1$ y $X + 0 = X$ indican que, cuando se realiza una operación OR con 1, la variable binaria X es 1 pero la OR con 0 deja la variable sin cambiar. De esta forma, un determinado bit o grupo de bits de un registro pueden ponerse a 1 si se hace una operación OR con 1. Considere el siguiente ejemplo:

10101101 10101011	$R1$	(dato)
11111111 00000000	$R2$	(máscara)
11111111 10101011	$R1 \leftarrow R1 \vee R2$	

El byte de mayor peso de $R1$ está todo a 1 al hacer la operación lógica OR con todos los unos de $R2$. El byte de menor peso permanece sin cambiar puesto que se ha hecho la operación lógica OR con ceros.

La microoperación XOR (OR exclusiva) puede usarse para complementar uno o más bits de un registro. Las ecuaciones booleanas $X \oplus 1 = \bar{X}$ y $X \oplus 0 = X$ determinan que la operación XOR de una variable X con 1 da como resultado el complemento de X pero con 0 la variable permanece sin cambiar. Haciendo la operación lógica XOR a un bit o grupo de bits del registro $R1$ con 1 en determinadas posiciones de $R2$, es posible complementar los bits de dichas posiciones de $R1$. Considere el siguiente ejemplo:

10101101 10101011	$R1$	(dato)
11111111 00000000	$R2$	(máscara)
01010010 10101011	$R1 \leftarrow R1 \oplus R2$	

El byte de mayor peso de $R1$ se complementa después de realizar la operación XOR con $R2$, y el byte de menor peso permanece inalterado.

Microoperaciones de desplazamiento

Las microoperaciones de desplazamiento se usan para movimientos laterales de datos. El contenido del registro fuente puede desplazarse tanto a la derecha como a la izquierda. Un desplazamiento a la izquierda es hacia el bit más significativo, y un desplazamiento a la derecha es hacia el bit menos significativo. Las microoperaciones de desplazamiento se usan para transferencia de datos vía serie. También se usan para manipular el contenido de registros en operaciones aritméticas, lógicas y de control. En una microoperación de desplazamiento, el registro de destino puede ser el mismo o diferente del registro origen. Para representar las microoperaciones definidas en la Tabla 7-5 usamos una serie de cadenas de caracteres. Por ejemplo:

$$R0 \leftarrow \text{sr } R0,\ R1 \leftarrow \text{sl } R2$$

son dos microoperaciones que especifican un desplazamiento de un bit a la derecha del contenido del registro $R0$, sr, y la transferencia del contenido de $R2$ desplazado un bit a la izquierda al registro $R1$, sl, respectivamente. El contenido de $R2$ no se cambia el realizar el desplazamiento.

En una microoperación de desplazamiento a la izquierda, el bit más a la derecha decimos que es el *bit entrante*. En una microoperación de desplazamiento a la derecha, definimos al bit más a la izquierda como el bit entrante. El bit entrante puede tomar diferentes valores, depen-

□ **TABLA 7-5**
Ejemplos de desplazamientos

		Ejemplos de 8 bits	
Tipos	**Designación simbólica**	**Fuente *R2***	**Después de desplazar: Destino *R1***
Desplazamiento a la izquierda	$R1 \leftarrow \text{sl } R2$	10011110	00111100
Desplazamiento a la derecha	$R1 \leftarrow \text{sr } R2$	11100101	01110010

diendo del tipo de microoperación de desplazamiento. Aquí asumimos que para sr y sl el bit entrante es 0, como se muestra en los ejemplos de la Tabla 7-5. Para una operación de desplazamiento a la izquierda, el bit saliente es el más a la izquierda, y para una operación de desplazamiento a la derecha, es el más a la derecha. En estos desplazamientos, el bit saliente se desecha. En el Capítulo 11, exploraremos otros tipos de desplazamientos que difieren en el tratamiento de los bits entrantes y salientes

7-6 MICROOPERACIONES EN UN REGISTRO

Esta sección cubre la ejecución de una o más operaciones sobre un sólo registro como destino de los resultados principales. El registro también puede servir como fuente de un operando para hacer operaciones unarias y binarias. Debido a la relación tan estrecha en un conjunto de elementos de almacenamiento y las microoperaciones, se supone que la lógica combinacional forma parte del registro y la llamaremos *lógica dedicada* del registro. Por contra, está la lógica que se comparte por varios registros destino. En este caso, a esta lógica se le llama *lógica compartida* por los registros destino.

La lógica combinacional necesaria para realizar las microoperaciones, descrita en secciones precedentes, puede usar uno o más bloques funcionales de los Capítulos 4 y 5 o se puede diseñar especialmente para los registros. Inicialmente, se usarán bloques funcionales en combinación con flip-flops tipo *D* con o sin habilitación. Se presenta una técnica sencilla, usando multiplexores, que permite seleccionar múltiples microoperaciones en un registro sencillo. A continuación, a partir de dicho registro, se diseña un registro que puede desplazar y contar.

Transferencias basadas en multiplexores

Hay ocasiones en que un registro recibe datos de 2 o más fuentes diferentes en distintos instantes de tiempo. Considere la siguiente sentencia condicional que tiene la forma *if-then-else*:

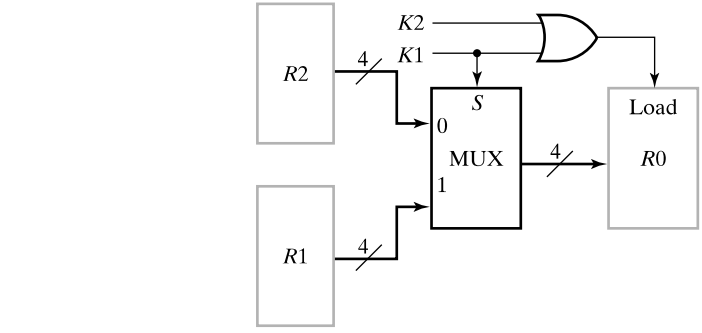
$$\text{if } (K_1 = 1) \text{ then } (R0 \leftarrow R1) \text{ else if } (K_2 = 1) \text{ then } (R0 \leftarrow R2)$$

El valor del registro $R1$ se transfiere al registro $R0$ cuando la señal de control K_1 es igual a 1. Cuando $K_1 = 0$, el valor del registro $R2$ se transfiere a $R0$ si K_2 es igual a 1. En el resto de los casos el contenido de $R0$ permanece sin alterar. La sentencia condicional puede ser dividida en dos partes usando las siguientes condiciones de control.

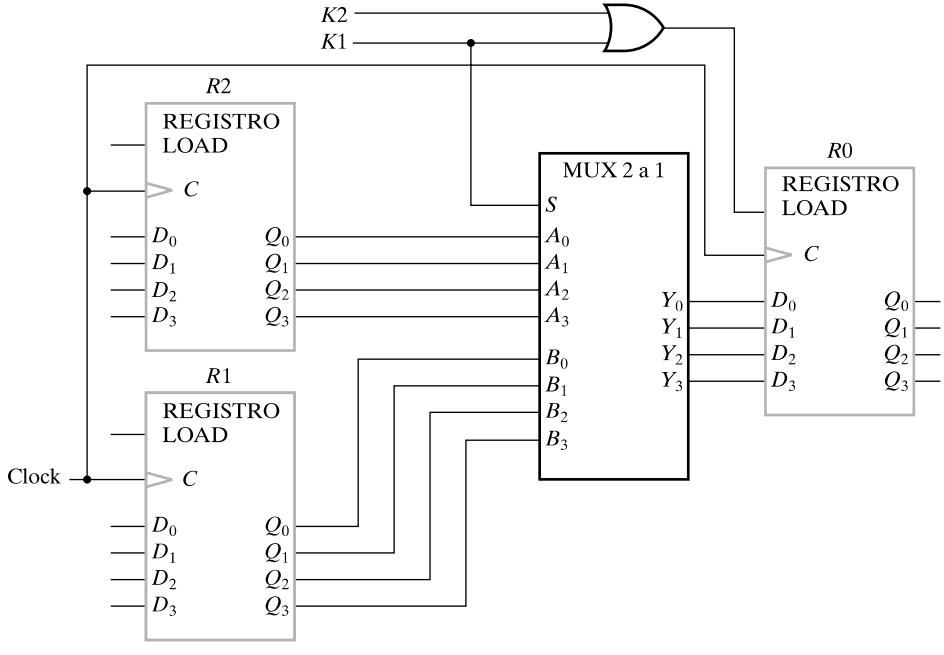
$$K_1\colon R0 \leftarrow R1,\ \bar{K}_1K_2\colon R0 \leftarrow R2$$

Esto especifica las conexiones hardware de dos registros, $R1$ y $R2$ a un registro común de destino, $R0$. Además, la selección de los dos registros fuentes debe basarse en los valores de las variables de control K_1 y K_2.

En la Figura 7-7(a) se muestra el diagrama de bloques de un circuito, con registros de 4 bits, que realiza la sentencia condicional de transferencia de registros usando un multiplexor. El multiplexor 2 a 1 selecciona entre dos registros fuente. Para $K_1 = 1$, $R1$ se carga en $R0$, independientemente del valor de K_2. Para $K_1 = 1$ y $K_2 = 1$, $R2$ se carga en $R0$. Si tanto K_1 como K_2 son iguales a 0, el multiplexor selecciona a $R2$ como entrada de $R0$, pero como la función de control, $K_2 + K_1$, conectada a la entrada de carga de $R0$, LOAD, es igual a 0, el contenido de $R0$ permanece sin cambiar.



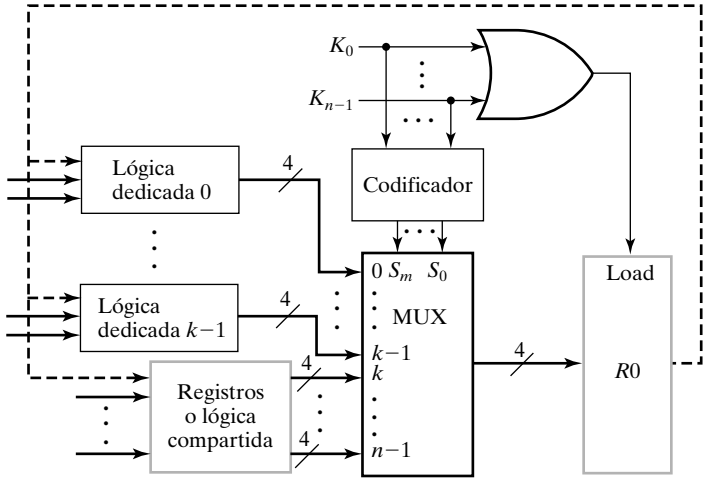
(a) Diagrama de bloques



(b) Lógica detallada

□ **FIGURA 7-7**
Uso de multiplexores entre dos registros

En la Figura 7-7(b) se muestra el diagrama lógico detallado para llevar a cabo el hardware. El diagrama usa símbolos funcionales, basados en los registros de la Figura 7-2, y un multiplexor, de buses de 4 bits, 2 a 1 del Capítulo 4. Véase que, puesto que el diagrama sólo representa parte de un sistema, hay entradas y salidas que aún no están conectadas. El reloj tampoco se muestra en el diagrama de bloques pero si en el diagrama detallado. Es importante relacionar la información dada en un diagrama de bloques, como el de la Figura 7-7(a), con las conexiones detalladas del diagrama lógico de la Figura 7-7(b). Con el motivo de ahorrar espacio, frecuentemente omitiremos el diagrama lógico detallado de los diseños. Sin embargo, es posible obtener un diagrama lógico detallado del conexionado a partir del correspondiente diagrama de bloques y una biblioteca de bloques funcionales. De hecho, tal procedimiento lo realizan los programas de computadora para síntesis lógica automatizada.



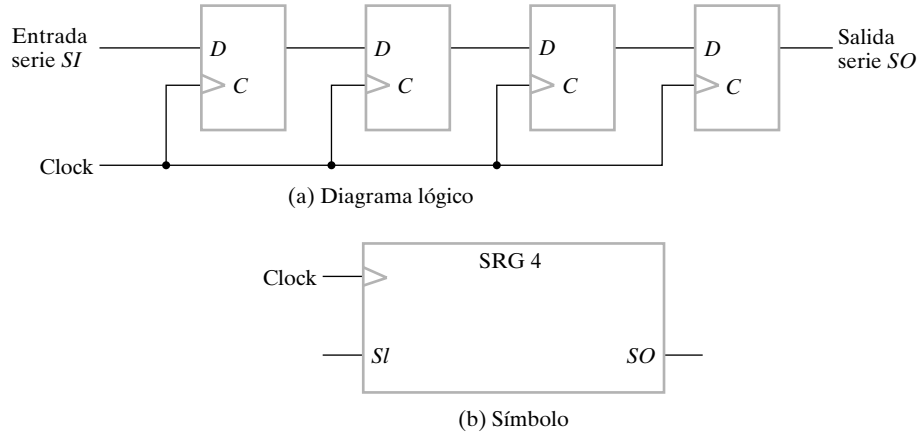
□ **FIGURA 7-8**
Selección con multiplexores generalizada para *n* fuentes

El ejemplo precedente puede generalizarse haciendo que el multiplexor tenga *n* fuentes y estas fuentes ser salidas registradas o lógica combinacional que realiza microoperaciones. De esta generalización resulta el diagrama mostrado en la Figura 7-8. El diagrama supone que cada fuente pueden ser tanto salidas de un registro como lógica combinacional que realiza una o más microoperaciones. En los casos en los que las microoperaciones se destinan a los registros, la lógica dedicada se incluye como parte del registro. En la Figura 7-8, las primeras *k* fuentes son de lógica dedicada y las últimas $n - k$ fuentes pueden ser tanto registros como lógica compartida. Las señales de control que seleccionan un determinado recurso son, o bien una sola variable de control o la operación OR de todas las señales correspondientes a las microoperaciones asociadas a los recursos involucrados. Para forzar una carga a *R*0 para una microoperación, se realiza una operación OR de estas señales de control para crear la señal de carga Load. Suponiendo que sólo una de las señales de control es 1 en cualquier instante de tiempo, estas señales deben ser codificadas para proporcionar los códigos de selección para el multiplexor. Son posibles dos modificaciones de la estructura dada. Las señales de control podrían aplicarse a un circuito $2 \times n$ AND-OR (es decir, un multiplexor con el decodificador eliminado). Alternativamente, las señales de control podrían ya estar codificadas, evitando el uso del código todo a ceros, de forma que la puerta OR genera la señal de carga correctamente.

Registros de desplazamiento

Un registro de desplazamiento es capaz de desplazar sus bits almacenados lateralmente en una o ambas direcciones. La configuración lógica de un registro de desplazamiento consiste en una cadena de flip-flops, con la salida de un flip-flop conectada a la entrada del siguiente flip-flop. Todos los flip-flops tienen una entrada común de reloj que activa el desplazamiento.

El registro más simple sólo usa flip-flops, como se muestra en la Figura 7-9(a). La salida se conecta a la entrada *D* de los flip-flops a su derecha. El reloj es común a todos los flip-flops. La *entrada serie*, *SI*, es la entrada al flip-flop más a la izquierda. La salida *serie*, *SO*, se toma de la salida del flip-flop más a la derecha. En la Figura 7-9(b) se ofrece un símbolo para representar al registro de desplazamiento.



□ **FIGURA 7-9**
Registro de desplazamiento de 4 bit

A veces es necesario controlar el registro de forma que sólo desplace en los flancos positivos deseados del reloj. En el registro de la Figura 7-9, el desplazamiento se puede controlar en el reloj, usando la lógica mostrada en la Figura 7-1(c), con la señal Shift reemplazando a la de Load. De nuevo, debido al *skew* de reloj, no es una solución deseable. Más adelante aprenderemos que las operaciones de desplazamiento se pueden controlar mediante las entradas *D* de los flip-flops mejor que a través de las entradas de reloj *C*.

REGISTRO DE DESPLAZAMIENTO CON CARGA EN PARALELO Si todas las salidas del registro son accesibles, la información introducida vía serie, mediante desplazamientos, puede accederse en paralelo de las salidas de los flip-flops. Si se añade la capacidad de carga en paralelo al registro de desplazamiento, entonces el dato cargado en paralelo puede desplazarse saliendo vía serie. Así, un registro con las salidas de los flip-flops accesibles y carga en paralelo puede utilizarse para convertir el dato paralelo que entra en un dato serie que sale y viceversa.

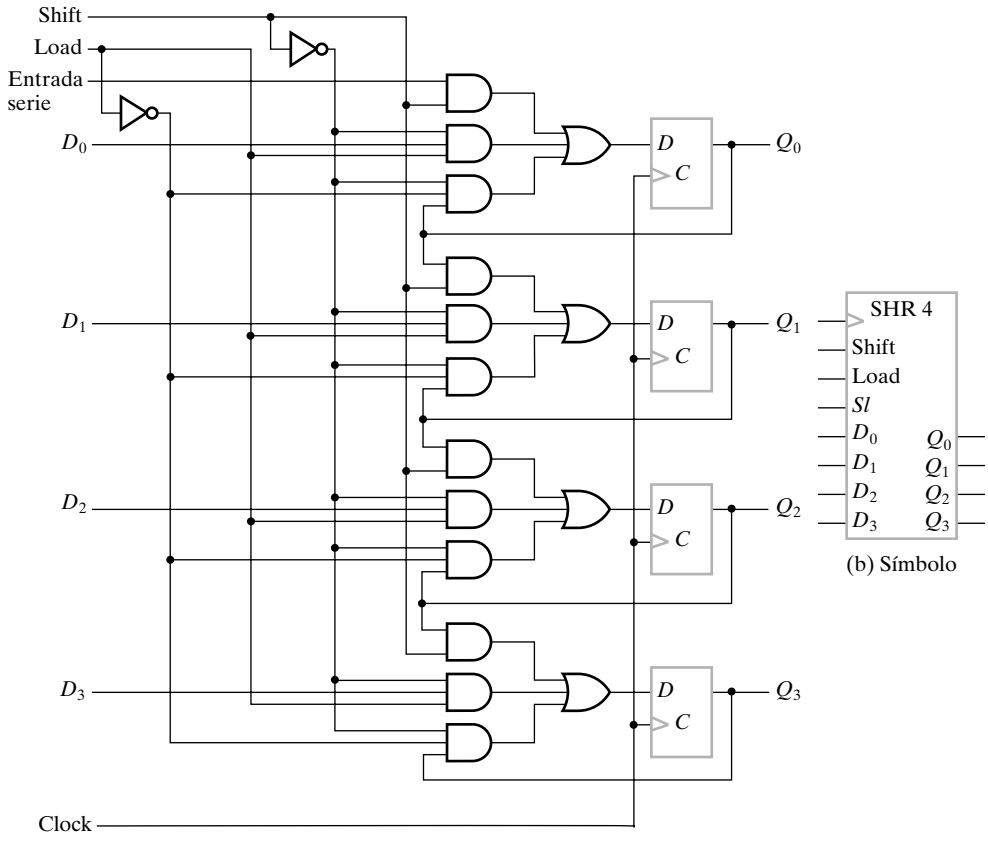
En la Figura 7-10 se muestra el diagrama lógico y el símbolo de un registro de desplazamiento de 4 bits con carga en paralelo. Existen dos entradas de control, una para el desplazamiento y la otra para la carga. Cada etapa del registro se compone de un flip-flop tipo *D*, una puerta OR, y tres puertas AND. La primera AND permite la operación de desplazamiento. La segunda puerta AND habilita la entrada de datos. La tercera puerta AND repone el contenido del registro cuando no se solicita ninguna operación.

En la Tabla 7-6 se especifica la forma de operar de este registro y, además, se da mediante transferencia de registros:

$$Shift\colon\ Q \leftarrow \mathrm{sl}Q$$

$$\overline{Shift} \cdot Load\colon\ Q \leftarrow D$$

La operación «no cambio» está implícita si ninguna de las condiciones de transferencia se satisface. Cuando ambas entradas de control, Shift y Load, son 0, la tercera puerta de cada etapa está habilitada y la salida de cada flip-flop se aplica a su propia entrada *D*. Un flanco positivo de reloj repone el contenido del registro y la salida no cambia. Cuando la entrada Shift es 0 y Load es 1, la segunda puerta AND de cada etapa está habilitada y la entrada D_i se aplica a la entrada *D* del flip-flop correspondiente. El siguiente flanco de subida transfiere la entrada paralelo al



□ **FIGURA 7-10**
Registro de desplazamiento con carga en paralelo

□ **TABLA 7-6**
Tabla de funcionamiento del registro de la Figura 7-10

Desplazamiento	Carga	Operación
0	0	No cambia
0	1	Carga paralela de datos
1	×	Desplaza de Q_0 a Q_3

registro. Cuando la entrada Shift es igual a 1, la primera puerta AND de cada etapa se habilita y las otras dos se deshabilitan. Puesto que la entrada Load esta deshabilitada por la entrada Shift en la segunda puerta AND, la marcamos con una condición indiferente en la fila de Shift de la tabla. Cuando llega un flanco de reloj, la operación de desplazamiento hace que el dato de la entrada serie, *SI*, se transfiera al flip-flop Q_0, la salida Q_0 se transfiera al flip-flop Q_1, y así sucesivamente. Véase que debido a cómo está dibujado el circuito, la transferencia se realiza hacía abajo. Si rotamos la página un cuarto de vuelta en sentido contrario a las agujas de reloj, el registro desplaza de izquierda a derecha.

Los registros de desplazamiento se usan frecuentemente en interfaces entre sistemas digitales lejanos unos de otros. Por ejemplo, suponga que es necesario transmitir una cantidad de *n* bit entre dos puntos. Si la distancia es grande, será caro utilizar *n* líneas para transmitir *n* bits en paralelo. Puede ser más económico usar una única línea y transmitir la información vía serie, un bit en cada instante. El transmisor carga los *n* bits en paralelo en el registro de desplazamiento y luego transmite los datos vía serie a lo largo de la línea común. El receptor acepta los datos en serie en un registro de desplazamiento. Cuando los *n* bit se han acumulado, se pueden coger en paralelo de la salida del registro. Así, el transmisor hace una conversión paralelo-serie de los datos y el receptor hace una conversión serie-paralelo.

REGISTRO DE DESPLAZAMIENTO BIDIRECCIONAL Un registro capaz de desplazar en una sola dirección se llama *registro de desplazamiento unidireccional*. Un registro que puede desplazar en ambas direcciones se llama *registro de desplazamiento bidireccional*. Es posible modificar el circuito de la Figura 7-10 añadiendo una cuarta puerta AND en cada etapa, para desplazar el dato en dirección ascendente. Examinado el circuito resultante revelará que, las cuatro puertas AND junto con la puerta OR de cada etapa, constituye un multiplexor con las entradas de selección controlando la operación del registro.

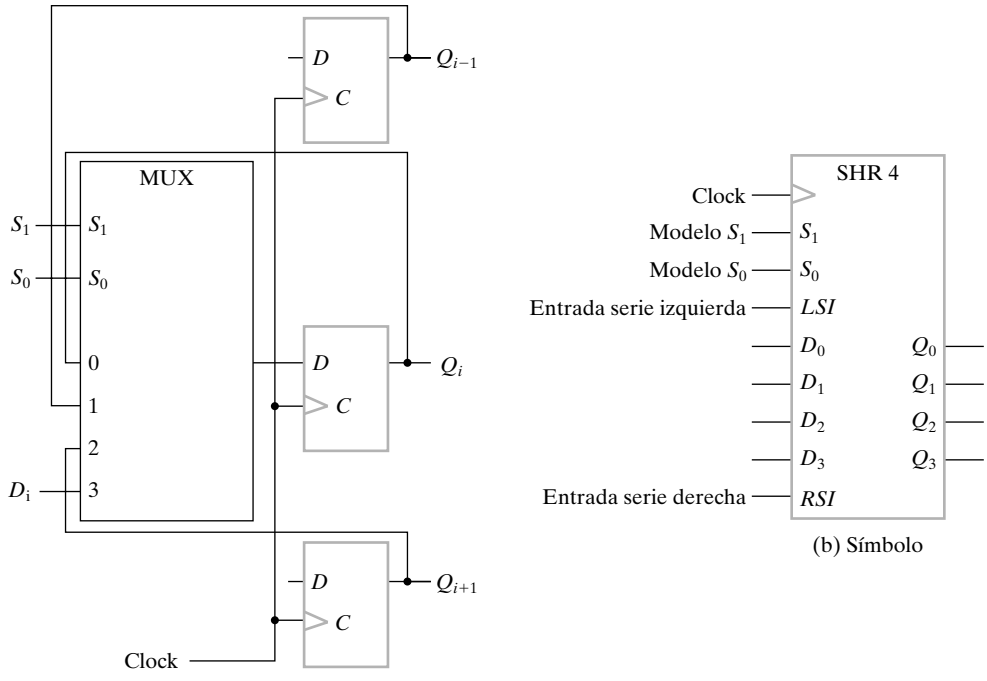
En la Figura 7-11(a) se muestra una etapa de un registro de desplazamiento bidireccional con carga en paralelo. Cada etapa está compuesta por un flip-flop tipo *D* y un multiplexor 4 a 1. Las 2 entradas de selección S_1 y S_0 seleccionan una de las entradas del multiplexor que se aplica al flip-flop tipo *D*. Las líneas de selección controlan el modo de operación del registro de acuerdo con la tabla de funcionamiento de la Tabla 7-7 y la siguiente transferencia de registros:

$$\bar{S}_1 \cdot S_0 = Q \leftarrow \text{sl}Q$$
$$S_1 \cdot \bar{S}_0 = Q \leftarrow \text{sr}Q$$
$$S_1 \cdot S_0 = Q \leftarrow D$$

□ **TABLA 7-7**
Tabla de funcionamiento del registro de la Figura 7-7

Modo de control		Operaciones del registro
S_1	S_0	
0	0	No cambia
0	1	Desplaza hacia abajo
1	0	Desplaza hacia arriba
1	1	Carga paralela

La operación «no cambia» está implícita si ninguna de las condiciones de transferencia se satisface. Si el modo de control es $S_1S_0 = 00$, se selecciona la entrada 0. Esto forma un camino desde la salida de cada flip-flop a su propia entrada. En el siguiente flanco de reloj se transfiere el valor en curso almacenado de nuevo a cada flip-flop y no ocurre cambio de estado. Si $S_1S_0 = 01$, el terminal marcado con 1 en el multiplexor tiene un camino hasta la entrada *D* de cada flip-flop. Este camino facilita la operación de desplazamiento hacia abajo. La entrada serie se transfiere a la primera etapa y el contenido de la etapa Q_{i-1} se transfiere a la etapa Q_i. Cuando $S_1S_0 = 10$, se produce una operación de desplazamiento hacia arriba sobre una segunda en-



(a) Diagrama lógico típico de una etapa

(b) Símbolo

□ **FIGURA 7-11**
Registro de desplazamiento bidireccional con carga en paralelo

trada serie que entra en la última etapa. Además, el valor de la etapa Q_{i+1} se transfiere a la etapa Q_i. Finalmente, si $S_1S_0 = 11$, la información binaria de cada línea de entrada paralelo se transfiere al flip-flop correspondiente, dando lugar a una carga en paralelo.

En la Figura 7-11(b) se muestra el símbolo del registro de desplazamiento bidireccional de la Figura 7-11(a). Véase que se le añaden tanto una entrada serie izquierda (*LSI*) como una entrada serie derecha (*RSI*). Si se desea tener salida serie, Q_3 se usa para desplazar a la izquierda y Q_0 para la derecha.

Contador asíncrono

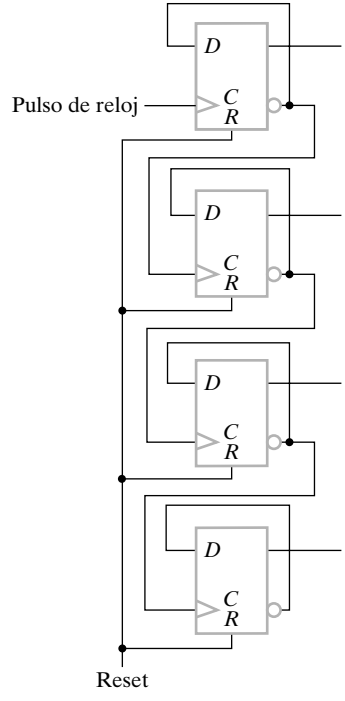
A un registro que pasa a través de una secuencia prescrita de distintos estados, según se aplica una secuencia de pulsos de entrada, se le llama contador. Los pulsos de entrada pueden ser pulsos de reloj u originarse desde otra fuente, y pueden darse lugar de forma periódica o no. En nuestra discusión sobre los contadores, suponemos que se usan pulsos de reloj aunque otras señales pueden sustituir al reloj. La secuencia de estados puede seguir la secuencia natural de los números binarios u otra secuencia de estados definida. Un contador que sigue la secuencia natural de números binarios se llama contador binario. Un contador binario de *n* bits está compuesto de *n* flip-flops y puede contar en binario desde 0 hasta $2^n - 1$.

Los contadores se pueden dividir en dos categorías: contadores asíncronos (en inglés *ripple counters*) y contadores síncronos. En un contador asíncrono, las transiciones de las salidas de los flip-flops sirven como fuente para disparar los cambios en los otros flip-flops. En otras palabras, las entradas *C* de algunos flip-flops no se disparan por un pulso común de reloj, sino por

transiciones que suceden en las salidas de otros flip-flops. En un contador síncrono, las entradas *C* de todos los flip-flops reciben un pulso de reloj simultáneamente, y el cambio de estado se determina a partir del estado presente del contador. Los contadores síncronos se discuten en las siguientes dos subsecciones. Aquí presentamos el contador asíncrono y se explica su funcionamiento.

En la Figura 7-12 se presenta el diagrama lógico de un contador asíncrono de 4 bits. El contador se construye a partir de flip-flops tipo *D* conectados de tal forma que la aplicación de un flanco positivo a la entrada *C* de cada flip-flop hace que se complemente su estado. La salida complementada de cada flip-flop se conecta a la entrada *C* del siguiente flip-flop más significativo. El flip-flop que contiene el bit menos significativo recibe los pulsos que llegan del reloj. El disparo mediante el flanco positivo hace que cada flip-flop complemente su valor cuando la señal en su entrada *C* recibe un flanco positivo. Esta transición positiva ocurre cuando la salida complementada del flip-flop anterior, que se conecta a la entrada *C*, bascula de 0 a 1. Un nivel alto en la señal de Reset, que maneja las entradas *R*, pone a cero a todos los registros asíncronamente.

Para comprender la forma de operar de un contador asíncrono vamos a examinar la secuencia de cuenta ascendente dada en la parte izquierda de la Tabla 7-8. La cuenta comienza en 0 y se incrementa en uno con cada pulso de conteo. Después de la cuenta 15, el contador regresa a 0 y repite la cuenta. El bit menos significativo (Q_0) se complementa con cada pulso de reloj. Cada vez que Q_0 va de 1 a 0, $\overline{Q_0}$ va de 0 a 1, complementando a Q_1. Cada vez que Q_1 bascula de 1 a 0, se complementa Q_2. Cada vez que Q_2 cambia de 1 a 0 se complementa Q_3, y así sucesivamente para los bits de mayor peso del contador asíncrono. Por ejemplo, considere la transición 0011 a 0100. Q_0 se complementa con el flanco de subida del pulso de cuenta. Puesto que Q_0



□ **FIGURA 7-12**
Contador asíncrono de 4 bits

□ **TABLA 7-8**
Secuencia de cuenta de un contador binario

Secuencia de cuenta ascendente				Secuencia de cuenta descendente			
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	0
0	0	1	0	1	1	0	1
0	0	1	1	1	1	0	0
0	1	0	0	1	0	1	1
0	1	0	1	1	0	1	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	0	0
1	0	0	0	0	1	1	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	0	1
1	0	1	1	0	1	0	0
1	1	0	0	0	0	1	1
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	1
1	1	1	1	0	0	0	0

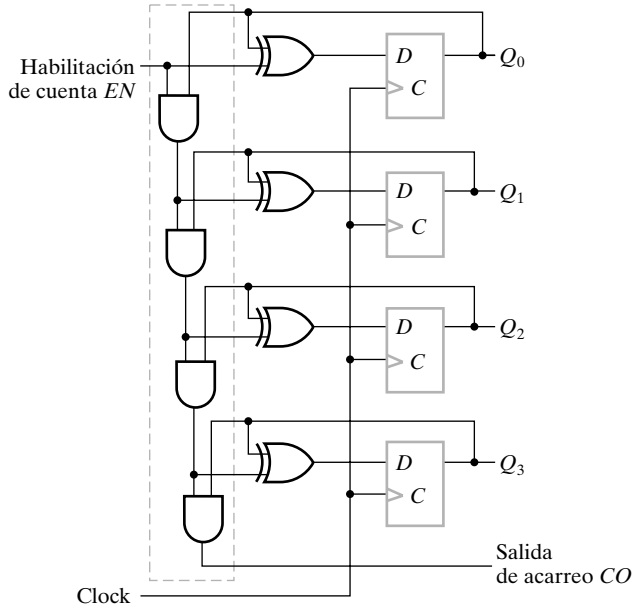
pasa de 1 a 0, dispara a Q_1 y se complementa. Como resultado, Q_1 bascula de 1 a 0, que complementa a Q_2, cambiando de 0 a 1. Q_2 no dispara a Q_3, puesto que $\overline{Q_2}$ produce una transición negativa y el flip-flop sólo responde a transiciones positivas. De esta forma se consigue la cuenta 0011 a 0100, cambiando los bits cada uno en un instante distinto de tiempo. El contador va de 0011 a 0010 (Q_0 de 1 a 0), luego a 0000 (Q_1 de 1 a 0) y finalmente a 0100 (Q_2 de 0 a 1). Cada flip-flop cambia en un instante de tiempo distinto en una rápida sucesión como si la señal se propagase a través del contador formando una onda de una etapa a la otra.

El contador asíncrono que realiza una cuenta descendente sigue la secuencia dada en la parte izquierda de la Tabla 7-8. La cuenta descendente puede generarse conectando la salida no invertida de cada flip-flop a la entrada *C* del siguiente.

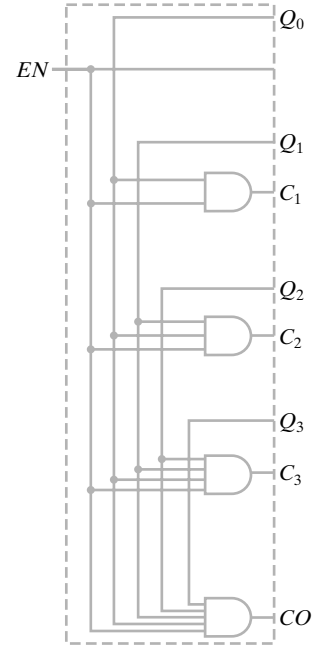
La ventaja del contador asíncrono es su sencillo hardware. Desafortunadamente, hay circuitos asíncronos, con lógica añadida que puede dar lugar a circuitos dependientes de retardos y de funcionamiento no fiable. Esto se cumple particularmente para la lógica que proporciona realimentaciones desde las salidas del contador hasta las entradas del mismo. Además, debido a la cantidad de tiempo que se necesita para terminar la onda, los contadores asíncronos grandes pueden ralentizar el circuito. Como consecuencia de todo esto, se favorece el uso de contadores binarios síncronos en todo tipo de diseño aunque en diseño de bajo consumo tienen una ventaja (*véase* el Problema 7-11).

Contadores binarios síncronos

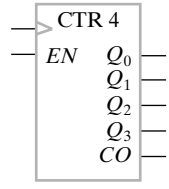
Los contadores síncronos, en contraste con los contadores asíncronos, tienen el reloj aplicado a las entradas *C* de todos los flip-flops. De esta forma, el reloj dispara a todos los flip-flops simultáneamente en vez de uno cada vez, como en un contador asíncrono. Un contador síncrono que



(a) Diagrama lógico con puertas serie



(b) Diagrama lógico con puertas en paralelo



(c) Símbolo

□ **FIGURA 7-13**
Contador síncrono binario de 4 bits

cuenta ascendentemente incrementando de 1 en 1 se puede construir a partir del incrementador de la Figura 5-12 y un flip-flop tipo *D*, según se muestra en la Figura 7-13(a). La salida de acarreo, *CO*, se añade sin colocar un valor X sobre la salida C_4 antes de la simplificación del sumador al incrementador de la Figura 5-12. La salida *CO* se usa para expandir el contador con más etapas.

Tenga en cuenta que los flip-flops se disparan con el flanco de subida del reloj. La polaridad del reloj no es esencial aquí, como fue en el contador asíncrono. El contador síncrono se puede dispara con el flanco positivo o con el flanco negativo[2].

CONTADORES SERIE Y PARALELO Usaremos el contador síncrono de la Figura 7-13 para demostrar dos alternativas de diseño de contadores binarios. En la Figura 7-13(a) se usa una cadena de puertas AND de 2 entradas para llevar la información de cada etapa a partir del estado de las anteriores etapas de contador. Esto es análogo a llevar la información lógica del acarreo ló-

[2] *N. del T.*: En un diseño digital real, generalmente no se encuentran mezclados flip-flops que se disparen con el flanco de subida con flip-flops que se disparan con el flanco de bajada.

gico de un sumador con acarreo serie. Un contador que usa tal lógica se dice que tiene *puertas en serie* y se le denomina *contador serie*. La analogía con el sumador con acarreo serie sugiere que pudiera haber un contador análogo a un sumador con acarro anticipado (en ingles *carry lookahead adder*). Tal lógica se puede extraer simplificando un sumador con acarreo anticipado, con el resultado que se muestra en la Figura 7-13(b). Esta lógica puede reemplazar sencillamente lo que aparece en la caja azul de la Figura 7-13(a) y dar lugar a un contador con *puertas en paralelo*, llamado *contador en paralelo*. La ventaja de las puertas en paralelo es que el paso del estado 1111 al 0000 sólo hay el retardo de una puerta AND en lugar del retardo de cuatro puertas AND que tiene el contador serie. Esta reducción del retardo permite a los contadores operar mucho más rápido.

Si conectamos 2 contadores de 4 bits juntos uniendo la salida *CO* de uno a la entrada EN del otro, el resultado es un contador de 8 bits serie-paralelo. Este contador tiene 2 partes de 4 bits en paralelo conectados en serie uno con otro. La idea puede extenderse a contadores de cualquier longitud. De nuevo, empleando la analogía con lo sumadores de acarreo anticipado, se pueden introducir niveles adicionales de puertas lógicas para reemplazar la conexión serie entre segmentos de 4 bits. La reducción adicional del retardo es útil para construir contadores grandes y rápidos.

En la Figura 7-13(c) se muestra el símbolo de un contador de 4 bits disparado con flanco positivo.

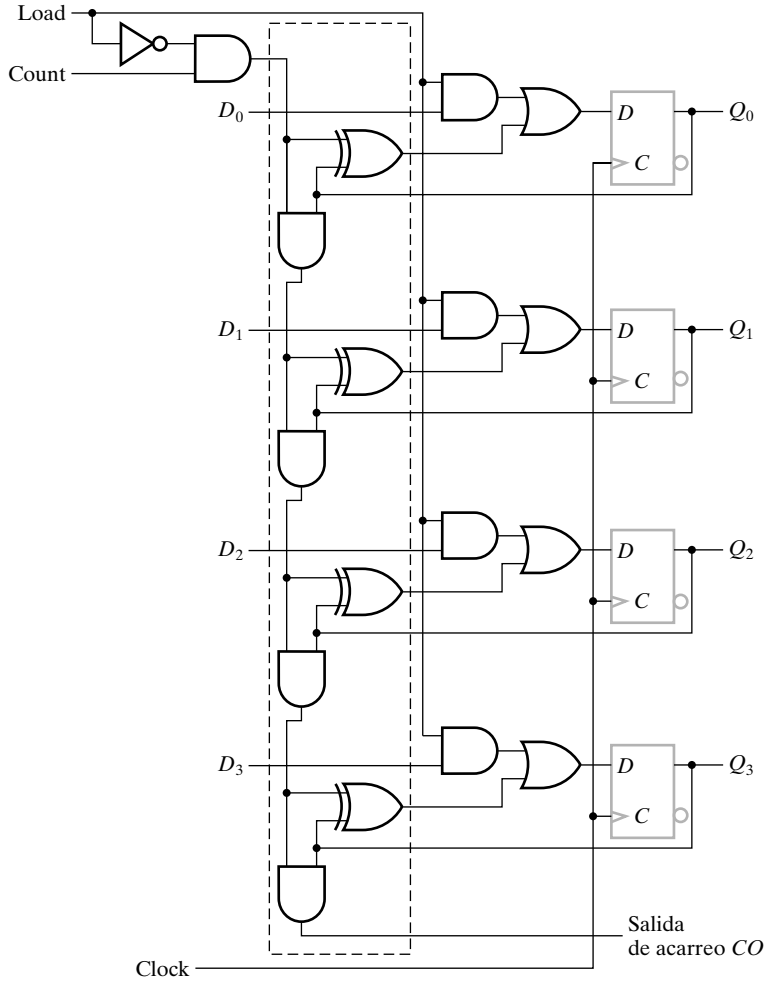
CONTADOR BINARIO ASCENDENTE-DESCENDENTE Un contador binario síncrono descendente pasa por sus estados en orden inverso desde 1111 hasta 0000 y vuelta al estado 1111 para repetir la cuenta. El diagrama lógico de un contador binario descendente es similar al circuito del contador ascendente, excepto que usa un decrementador en lugar de un incrementador. Las dos operaciones se pueden combinar para construir un contador que pueda contar tanto ascendente como descendentemente y al cual se le denomina contador binario ascendente-descendente. Tal contador se puede diseñar simplificando el sumador-restador de la Figura 5-8 a un incrementador-decrementador y añadiendo flip-flops tipo *D*. Cuenta ascendentemente para $S = 0$ y descendentemente para $S = 1$.

Alternativamente, se puede diseñar directamente un contador ascendente-descendente con habilitación a partir de un contador. Se necesita una entrada de modo para seleccionar entre las dos operaciones. Diseñamos esta entrada de selección de modo mediante *S*, $S = 0$ para cuenta ascendente y $S = 1$ para cuenta descendente. La variable *EN* es una entrada de habilitación de cuenta, con $EN = 1$ para contar tanto ascendente como descendentemente, y $EN = 0$ para deshabilitar ambas cuentas. Un contador ascendente-descendente de 4 bits se puede describir mediante las siguientes ecuaciones de entrada:

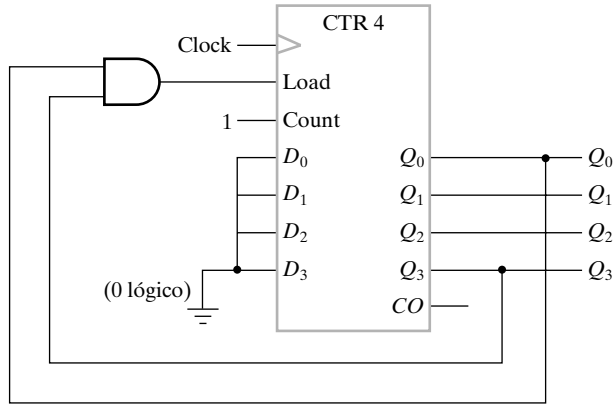
$$D_{A0} = Q_0 \oplus EN$$
$$D_{A1} = Q_1 \oplus ((Q_0 \cdot \bar{S} + \bar{Q}_0 \cdot S) \cdot EN)$$
$$D_{A2} = Q_2 \oplus ((Q_0 \cdot Q_1 \cdot \bar{S} + \bar{Q}_0 \cdot \bar{Q}_1 \cdot S) \cdot EN)$$
$$D_{A3} = Q_3 \oplus ((Q_0 \cdot Q_1 \cdot Q_2 \cdot \bar{S} + \bar{Q}_0 \cdot \bar{Q}_1 \cdot \bar{Q}_2 \cdot S) \cdot EN)$$

El diagrama lógico del circuito se puede obtener fácilmente de las ecuaciones de entrada, aunque no se incluye aquí. Se debería notar que las ecuaciones, tal y como están escritas, dan lugar a puertas en paralelo usando distinta lógica de acarreo para la cuenta ascendente y descendente. También es posible usar dos cadenas distintas de puertas en serie. Por contra, el contador extraído del incrementador-decrementador sólo usa una cadena de acarreo. En general, el coste lógico es similar.

CONTADOR BINARIO CON CARGA EN PARALELO Los contadores utilizados en los sistemas digitales necesitan con frecuencia la capacidad de realizar una carga en paralelo para transferir un número inicial al contador antes de contar. Esta función se puede realizar mediante un incrementador con habilitación, ENABLE, *n* ENABLEs y *n* puertas OR de 2 entradas, como se muestra en la Figura 7-14. Los *n* ENABLEs se usan para habilitar y deshabilitar la carga paralelo del dato de entrada, *D*, usando la señal Load. Véase que el ENABLE del incrementador se usa para habilitar o deshabilitar usando la expresión $\text{Count} \cdot \overline{\text{Load}}$. Si ambas entradas, Load y Count, son 0, las salidas no cambian, incluso cuando se aplican pulsos a las entradas *C*. Si la entrada de carga se mantiene a nivel 0, la entrada Count controla la operación del contador, y las salidas cambian a la siguiente cuenta binaria para cada flanco positivo del reloj. El dato puesto en las entradas *D* se carga en los flip-flops si Load es igual a 1, independientemente del valor de Count puesto que hace una operación AND con $\overline{\text{Load}}$ y Count. Los contadores con carga en paralelo son muy útiles en el diseño de procesadores digitales. En siguientes capítulos nos referiremos a ellos como registros con operaciones de carga e incremento.



□ **FIGURA 7-14**
Contador binario de 4 bits con carga en paralelo



□ **FIGURA 7-15**
Contador BCD

El contador binario con carga en paralelo se puede convertir en un contador BCD síncrono (sin entrada de carga) conectándole una puerta AND externa, como se muestra en la Figura 7-15. El contador comienza con todas sus salidas a cero, y la entrada de cuenta está activada siempre. Mientras la salida de la puerta AND sea 0, cada flanco de subida del reloj incrementa al contador en uno. Cuando las salidas alcancen la cuenta 1001, tanto Q_0 como Q_3 serán 1, haciendo que la salida de la puerta AND sea igual a 1. Esta condición hace que la entrada *Load* se active, y así, en el siguiente ciclo de reloj, el contador no cuenta pero se carga con el contenido de sus cuatro entradas *D*(3:0), que están conectadas a un 0 lógico, de esta forma se carga un 0000 después de la cuenta 1001. Así el circuito cuenta de 0000 hasta 1001, siguiéndole el 0000, como necesita un contador BCD.

Otros contadores

Un contador se puede diseñar para que genere cualquier número deseado de estados secuencialmente. Un *contador dividido por N* (también conocido como *contador módulo N*) es un contador que sigue una secuencia repetitiva de *N* estados. La secuencia puede seguir la cuenta binaria natural o cualquier otra secuencia arbitraria. En cualquier caso, el diseño de un contador sigue el procedimiento presentado en el Capítulo 6 para el diseño de circuitos secuenciales síncronos. Para demostrar este procedimiento, presentaremos el diseño de dos contadores: un contador BCD y un contador con una secuencia arbitraria de estados.

CONTADOR BCD Como se mostró en la sección anterior, un contador BCD se puede obtener a partir de un contador binario con carga paralela. También es posible diseñar un contador BCD directamente usando flip-flops y puertas por separado. Suponiendo que usamos flip-flops tipo *D* para el contador, presentamos en la Tabla 7-9 sus estados actuales y futuros. Se incluye, además, una salida *Y* en la tabla. Esta salida es igual a 1 si el estado actual es 1001, indicando el fin de la cuenta. De esta forma, la salida *Y* puede habilitar la cuenta de la siguiente década cuando su propia década cambia de 1001 a 0000.

Las ecuaciones para las entradas *D* se obtienen a partir del estado futuro de los valores listados en la tabla y se pueden simplificar mediante Mapas de Karnaugh. Los estados no utilizados

□ **TABLA 7-9**
Tabla de estados y entradas de los flip-flops para el contador BCD

Estado actual				Estado futuro				Salida
Q_8	Q_4	Q_2	Q_1	$D_8 = Q_8(t+1)$	$D_4 = Q_4(t+1)$	$D_2 = Q_2(t+1)$	$D_1 = Q_1(t+1)$	Y
0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	0	1	1	0
0	0	1	1	0	1	0	0	0
0	1	0	0	0	1	0	1	0
0	1	0	1	0	1	1	0	0
0	1	1	0	0	1	1	1	0
0	1	1	1	1	0	0	0	0
1	0	0	0	1	0	0	1	0
1	0	1	1	0	0	0	0	1

correspondientes a los miniterminos comprendidos entre 1010 y 1111, se usan con condiciones indiferentes. Las ecuaciones simplificadas del contador BCD son

$$D_1 = \bar{Q}_1$$
$$D_2 = Q_2 \oplus Q_1\bar{Q}_8$$
$$D_4 = Q_4 \oplus Q_1Q_2$$
$$D_8 = Q_8 \oplus (Q_1Q_8 + Q_1Q_2Q_4)$$
$$Y = Q_1Q_8$$

Los contadores BCD síncronos pueden conectarse en cascada para formar contadores para números decimales de cualquier longitud. Las cascada se hace reemplazando D_1 con $D_1 = Q_1 \oplus Y$ donde Y procede del contador BCD anterior. Además, debe realizarse una operación AND con Y y los productos a la derecha de cada una de las puertas XOR de cada ecuación desde D_2 hasta D_8.

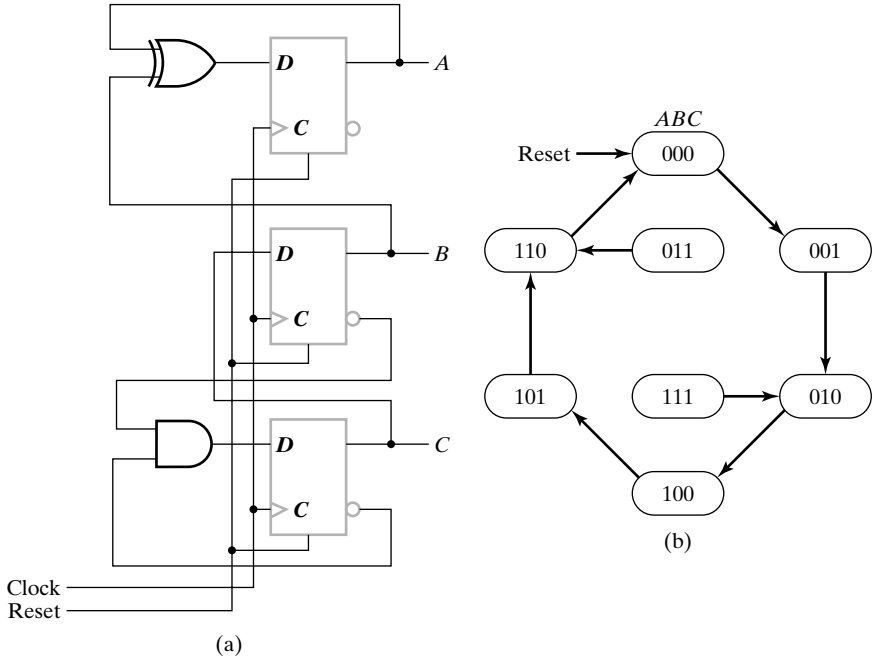
CONTADOR DE SECUENCIA ARBITRARIA Suponga que deseamos diseñar un contador que tiene una secuencia de seis estados que se repiten, como se muestra en la Tabla 7-10. En la secuencia, los flip-flops B y C repiten la cuenta binaria 00, 01, 10, mientras que el flip-flop A alterna entre 0 y 1 cada tres cuentas. Así que la secuencia del contador no es binaria natural y, además, hay dos estados, 011 y 111, que no están incluidos en la cuenta. Las ecuaciones de las entradas de los flip-flop D se pueden simplificar usando 3 miniterminos y 7 condiciones de indiferencia. Las funciones ya simplificadas son:

$$D_A = A \oplus B$$
$$D_B = C$$
$$D_C = \bar{B}\bar{C}$$

□ **TABLA 7-10**
Tabla de estados y entradas de los flip-flops para el contador

Estado actual			Estado futuro		
A	***B***	***C***	$DA = A(t+1)$	$DB = B(t+1)$	$DC = C(t+1)$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0

En la Figura 7-16(a) se muestra el diagrama lógico del contador. Puesto que hay dos estados no utilizados, analizamos el circuito para determinar su efecto. El diagrama de estados obtenido se dibuja en la Figura 7-16(b). Este diagrama indica que, si incluso el circuito va a alguno de estos estados sin utilizar, el siguiente pulso lo llevará a uno de los estados válidos y el circuito seguirá contando correctamente.



□ **FIGURA 7-16**
Contador con cuenta arbitraria

7-7 DISEÑO DE CÉLULAS BÁSICAS DE UN REGISTRO

En la Sección 5-1 discutimos los circuitos combinacionales iterativos o modulares que se pueden expandir fácilmente. En este capítulo conectamos tales circuitos a flip-flops para formar

circuitos secuenciales. Una célula de un bit, formada por un módulo combinacional conectado a un flip-flop, que hace que la salida sea la de un circuito secuencial con 2 estados, se le llama *célula básica de un registro*. Podemos diseñar un registro de *n* bit, con una o más microoperaciones asociadas, diseñando una célula de registro básica y haciendo *n* copias de ella. Dependiendo de si la salida del flip-flop es una entrada al circuito de la célula básica, la célula básica puede tener su estado futuro que dependa de su estado actual y de sus entradas o de sus entradas solamente. Si depende sólo de las entradas, se recomienda diseñar una célula básica combinacional y conectarla con el flip-flop de la célula secuencial básica a crear. Sin embargo, si el estado del flip-flop se realimenta a las entradas de la célula, se pueden seguir los métodos de diseño secuencial. En el siguiente ejemplo se diseña una célula básica para cada caso.

EJEMPLO 7-1 Diseño de células básicas de un registro

Un registro *A* debe realizar las siguientes operaciones de transferencias de registro:

AND: $A \leftarrow A \wedge B$
EXOR: $A \leftarrow A \oplus B$
OR: $A \leftarrow A \vee B$

Si no se especifica otra cosa, suponemos que

1. Sólo una de las variables de control AND, EXOR y OR es igual a 1.
2. Si AND, EXOR y OR son iguales 0, el contenido de *A* permanece sin cambiar.

Una forma sencilla de diseño de una célula de un registro que cumpla las condiciones 1 y 2 es utilizar el registro con carga paralelo construido con flip-flops tipo *D* con habilitación (Enable = Load) de la Figura 7-2. De esta forma, la expresión para la carga, LOAD, es la suma lógica de todas las señales de control que hacen que la transferencia ocurra. La expresión D_i es una suma de productos. Los términos que aparecen en cada producto son: una señal de control y la operación lógica correspondiente a esa señal de control.

En este ejemplo, las ecuaciones correspondientes para LOAD y D_i son:

$$\text{LOAD} = \text{AND} + \text{EXOR} + \text{OR}$$
$$D_i = A(t+1)_i = \text{AND} \cdot A_i B_i + \text{EXOR} \cdot (A_i \bar{B}_i + \bar{A}_i B_i) + \text{OR} \cdot (A_i + B_i)$$

La ecuación para D_i se hace de forma parecida a la utilizada en la parte de selección de un multiplexor en el que el conjunto de bloques de habilitación están conectados a una puerta OR. AND, EXOR y OR son las señales de habilitación, y la parte restante del producto es la función habilitada.

Usando los flip-flops tipo *D* para almacenar en el registro y sin utilizar la técnica de *clock gating*, se debe diseñar un multiplexor para cada célula:

$$D_{i,FF} = \text{LOAD} \cdot D_i + \overline{\text{LOAD}} \cdot A_i$$

La ecuación se da para mostrar la parte escondida dentro de la célula básica de un registro con carga en paralelo.

Una solución más compleja es diseñar directamente los flip-flops tipo *D* usando métodos de diseño de circuito secuenciales en lugar de soluciones *ad hoc* basadas en flip-flops con carga en paralelo.

Podemos formular una tabla de estados codificada con *A* como variable de estado y salida, y AND, EXOR, OR y B como entradas, como se muestra en la Tabla 7-11.

□ **TABLA 7-11**
Tabla de estados y entradas de flip-flops para contadores

Estado actual A	Estado futuro $A(t+1)$						
	(AND = 0) ·(EXOR = 0) ·(OR = 0)	**(OR = 1) ·(B = 0)**	**(OR = 1) ·(B = 1)**	**(EXOR = 1) ·(B = 0)**	**(EXOR = 1) ·(B = 1)**	**(AND = 1) ·(B = 0)**	**(AND = 1) ·(B = 1)**
0	0	0	1	0	1	0	0
1	1	1	1	1	0	0	1

Formulando la ecuación para la entrada del flip-flop $D_i = A(t+1)_i$,

$$D_i = A(t+1)_i = \text{AND} \cdot A_i \cdot B_i + \text{EXOR} \cdot (A_i\bar{B}_i + \bar{A}_iB_i) + \text{OR} \cdot (A_i + B_i) + \overline{\text{AND}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{OR}} \cdot A_i$$

Debido a la relación entre el operador OR y los operadores AND y EXOR y utilizando simplificaciones algebraicas, se consigue la siguiente expresión:

$$A(t+1)_i = (\text{OR} + \text{AND}) \cdot A_i \cdot B_i + (\text{OR} + \text{EXOR}) \cdot (A_i\bar{B}_i + \bar{A}_iB_i) + \overline{\text{AND} + \text{EXOR}} \cdot A_i$$

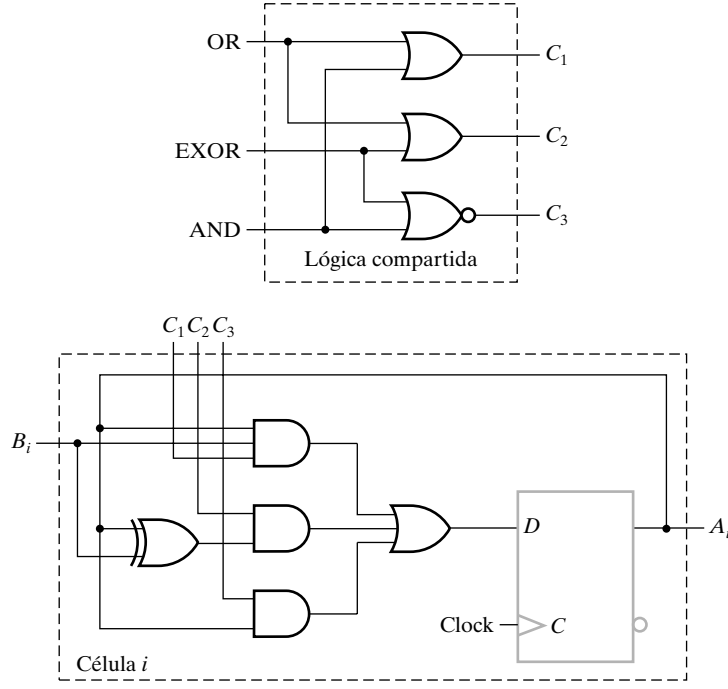
Los términos OR + AND, OR + EXOR, y $\overline{\text{AND} + \text{EXOR}}$ no dependen de los valores A_i y B_i asociados a cada una de las células. La lógica para estos términos puede ser compartida por todas las células del registro. Usando C_1, C_2 y C_3 como variables intermedias, resulta el siguiente conjunto de ecuaciones:

$$C_1 = \text{OR} + \text{AND}$$
$$C_2 = \text{OR} + \text{EXOR}$$
$$C_3 = \overline{\text{AND} + \text{EXOR}}$$
$$D_i = A(t+1)_i = C_1A_iB_i + C_2(A_i\bar{B}_i + \bar{A}_iB_i) + C_3 \cdot A_i$$

En la Figura 7-17 se muestra el diseño de la célula básica del registro y de la lógica compartida por todas las células del registro A. Antes de comparar estos resultados con los obtenidos con el método anterior, podemos aplicar una simplificación similar y compartir lógica a los resultados del método sencillo:

$$C_1 = \text{OR} + \text{AND}$$
$$C_2 = \text{OR} + \text{EXOR}$$
$$D_i = A(t+1)_i = C_1A_iB_i + C_2(A_i\bar{B}_i + \bar{A}_iB_i)$$
$$\text{LOAD} = C_1 + C_2$$
$$D_{i,FF} = \text{LOAD} \cdot D_i + \overline{\text{LOAD}} \cdot A_i$$

Si estas ecuaciones se usan directamente, el coste del método simple es algo más alto. Sin embargo, si estas ecuaciones se proporcionan a una herramienta de minimización en lugar de ser usadas directamente, resultarán las mismas ecuaciones que en el método complejo. De esta forma, el uso del método más sencillo no aumenta necesariamente el coste del hardware.



□ **FIGURA 7-17**
Diagrama lógico de la célula básica del registro diseñada en el Ejemplo 7-1 ■

En el ejemplo anterior, no hay conexiones laterales entre células adyacentes. Entre las operaciones que requieren conexiones laterales están los desplazamientos, operaciones aritméticas y comparaciones. Un método para el diseño de estas estructuras es combinar los diseños combinacionales dados en el Capítulo 5 con lógica de selección y flip-flops. En la Figura 7-8 se muestra un método general para hacer registros multifunciones usando flip-flops con carga en paralelo. Este sencillo método evita el diseño de células básicas de registros pero, si se lleva a cabo directamente, puede dar lugar a un exceso de lógica y demasiadas conexiones laterales. La alternativa es hacer una célula básica del registro a medida. En tales diseños, un factor crítico es la definición de las conexiones laterales necesarias. Las diferentes operaciones pueden ser definidas controlando la entrada de la célula menos significativa de la cascada de células. El método de diseño a medida se ilustra en el siguiente ejemplo diseñando una célula básica de un registro multifunción.

EJEMPLO 7-2 Diseño de células básicas de un registro

Un registro A debe realizar las siguientes transferencias de registros:

SHL: $A \leftarrow \text{sl}A$
EXOR: $A \leftarrow A \oplus B$
ADD: $A \leftarrow A + B$

Si no se especifica otra cosa, suponemos que:

1. Sólo una de las variables de control SHL, EXOR y ADD es igual a 1.
2. Si SHL, EXOR y ADD son iguales 0, el contenido de A permanece sin cambiar.

Un método sencillo de diseñar una célula básica de un registro que cumpla las condiciones 1 y 2 es usar un registro con carga en paralelo controlado por LOAD. En este método la expresión LOAD es la suma lógica de todas las señales de control que hacen que ocurra una transferencia. La realización para D_i consiste en una suma de productos, donde cada producto tiene una señal de control y la lógica para la operación.

Para este ejemplo, el resultado para las ecuaciones LOAD y D_i son:

$$\text{LOAD} = \text{SHL} + \text{EXOR} + \text{ADD}$$

$$D_i = A(t+1)_i = \text{SHL} \cdot A_{i-1} + \text{EXOR} \cdot (A_i \oplus B_i) + \text{ADD} \cdot ((A_i \oplus B_i) \oplus C_i)$$

$$C_{i+1} = (A_i \oplus B_i)C_i + A_iB_i$$

Estas ecuaciones pueden usarse sin modificación o pueden ser optimizadas.

Supongamos ahora que hacemos un diseño a medida suponiendo que todas las células del registro son idénticas. Esto significa que las células más y menos significativas son las mismas que las células del interior de la cadena. Por esto, el valor de C_0 debe especificarse y el uso de C_n, si existe, debe determinarse para cada una de las tres operaciones. Para el desplazamiento a la izquierda, suponemos que el bit más a la derecha vacante se rellena con un 0, haciendo $C_0 = 0$. Puesto que C_0 no está involucrado en la operación EXOR, se puede suponer indiferente. Finalmente, para la suma, C_0 puede suponerse 0 o puede dejarse como variable para permitir el acarreo de una suma anterior. Suponemos que C_0 es igual a 0 para la suma puesto que no se ha especificado un acarreo adicional en la sentencia de transferencia de registros.

Nuestro primer objetivo es minimizar las conexiones laterales entre células. Dos de las tres operaciones, desplazamiento a izquierda y la suma, necesitan una conexión lateral a la izquierda (es decir, hacia la célula final más significativa de la cadena de células). Nuestro objetivo es usar una señal para ambas operaciones, llamémosla C_i. Ya existe para la suma pero debe ser redefinida para efectuar la suma y el desplazamiento a la izquierda. En nuestro diseño a medida también se reemplaza el flip-flop con carga en paralelo por un flip-flop tipo *D*. Ahora formulamos la tabla de estados para la célula básica del registro en la Tabla 7-12:

$$D_i = A(t+1)_i = \overline{\text{SHL}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{ADD}} \cdot A_i + \text{SHL} \cdot C_i + \text{EXOR} \cdot (A_i \oplus B_i) + \text{ADD} \cdot (A_i \oplus B_i \oplus C_i)$$

$$C_{i+1} = \text{SHL} \cdot A_i + \text{ADD} \cdot (\cdot (A_i \oplus B_i)C_i + A_iB_i)$$

El término $A_i \oplus B_i$ aparece tanto en el término de EXOR como en el término ADD. De hecho, si $C_i = 0$ durante la selección de la operación EXOR, las funciones para la suma en ADD y para

□ **TABLA 7-12**
Tabla de estados y las entradas de los flip-flop para el diseño de la célula del registro del Ejemplo 7-2

Estado actual A_i	Entradas	Estado futuro $A_i(t+1)$/salida C_{i+1}			
	SHL = 0 **EXOR = 0** **ADD = 0**	**SHL = 1** B_i = 0 C_i = 0	1 0 1	1 1 0	1 1 1
		EXOR = 1 B_i = 0	1 1		
		ADD = 1 B_i = 0 C_i = 0	1 0 1	1 1 0	1 1 1
0	0/X	0/0	0/0	1/0	0/0	1/0	0/X	1/X	0/0	1/0	1/0	0/1
1	1/X	0/1	0/1	1/1	0/1	1/1	1/X	0/X	1/0	0/1	0/1	1/1

EXOR pueden ser idénticas. En la ecuación C_{i+1}, como SHL y ADD son 0 cuando EXOR es 1, C_i es 0 para todas las células de la cascada excepto para la menos significativa. De esta forma, los valores de entrada C_i son 0 para todas las células en el registro A. Así podemos combinar las operaciones ADD y EXOR como sigue:

$$D_i = A(t+1)_i = \overline{\text{SHL}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{ADD}} \cdot A_i + \text{SHL} \cdot C_i + (\text{EXOR} + \text{ADD}) \cdot ((A_i \oplus B_i) \oplus C_i)$$

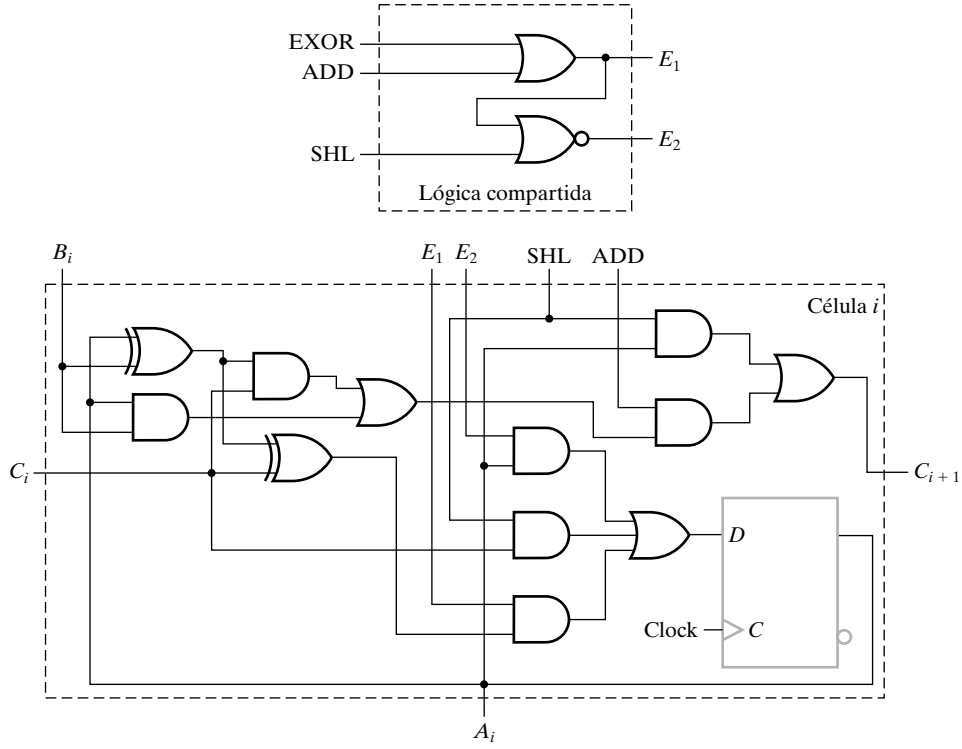
Las expresiones $\overline{\text{SHL}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{ADD}}$ y EXOR + ADD, que son independientes de A_i, B_i y C_i, pueden ser compartidas por todas las células. Las ecuaciones resultantes son:

$$E_1 = \text{EXOR} + \text{ADD}$$
$$E_2 = \overline{E_1 + \text{SHL}}$$
$$D_i = E_2 \cdot A_i + \text{SHL} \cdot C_i + E_1 \cdot ((A_i \oplus B_i) \oplus C_i)$$
$$C_{i+1} = \text{SHL} \cdot A_i + \text{ADD} \cdot ((A_i \oplus B_i)C_i + A_i B_i)$$

El registro resultante aparece en la Figura 7-18. Comparando este resultado con la célula básica del registro del diseño sencillo, notamos dos diferencias:

1. Sólo existe una conexión lateral entre células en lugar de dos.
2. La lógica se ha compartido muy eficientemente para las operaciones de suma y EXOR.

El diseño a medida ha ahorrado conexión y lógica no presentes a nivel de diseño de bloques con o sin optimización.

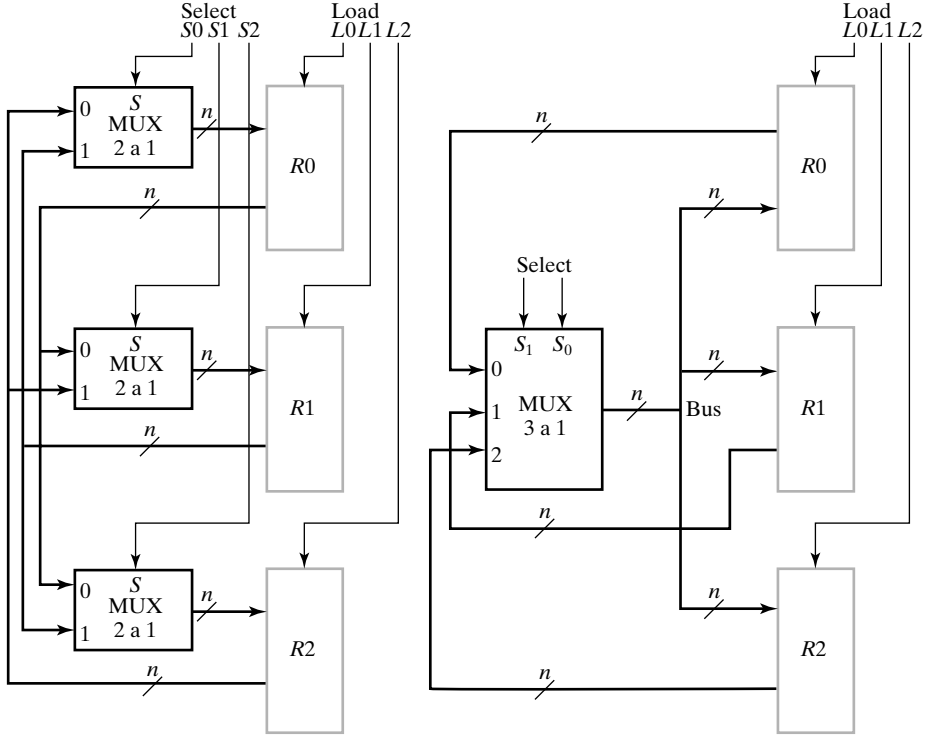


□ **FIGURA 7-18**
Diagrama lógico de la célula básica del registro diseñada en el Ejemplo 7-2 ■

7-8 TRANSFERENCIA DE MÚLTIPLES REGISTROS BASADA EN BUSES Y MULTIPLEXORES

Un sistema digital típico tiene muchos registros. Las rutas deben crearse para transferir datos de un registro a otro. La cantidad de lógica y el número de interconexiones puede ser excesiva si cada registro tiene su propio conjunto de multiplexores dedicados. Un esquema más eficiente para transferir datos entre registros es un sistema que usa un camino de transferencia compartido llamado *bus*. Un bus se caracteriza por un conjunto de líneas comunes que maneja la lógica de selección. Las señales de control de la lógica seleccionan una sola fuente y uno o más destinos en varios ciclos de reloj para realizar una transferencia.

En la Sección 7-4 vimos que los multiplexores y los registros con carga en paralelo se pueden usar para realizar transferencia de múltiples fuentes. En la Figura 7-19(a) se muestra un diagrama de bloques para dichas transferencias entre tres registros. Hay tres multiplexores, de buses de *n* bit, de 2 a 1, cada uno con su señal de selección. Cada registro tiene su propia señal de carga. El mismo sistema basado en un bus se puede llevar a cabo usando un sencillo multiplexor de buses de *n* bits de 3 a 1 y multiplexores y registros con carga en paralelo. Si se comparte un conjunto de salidas de multiplexores en un camino común, estas líneas de salida forman un bus. En la Figura 7-19(b) se muestra dicho sistema con un sólo bus para las transferencias a los tres registros. El par de líneas de control, Select, determina el contenido del registro fuente que aparecerá a las salidas del multiplexor (es decir, en el bus). Las entradas de carga, Load, determinan el registro o registros de destino que se cargan con el dato del bus.



(a) Multiplexores dedicados (b) Un solo bus

□ **FIGURA 7-19**
Bus simple versus multiplexores dedicados

En la Tabla 7-13 se ilustran las transferencias del diseño de la Figura 7-19(b) realizadas con un solo bus. La primera transferencia es de *R*2 a *R*0. Para *Select* igual a 10, se selecciona la entrada *R*2 del multiplexor. La señal de carga *L*0 del registro *R*0 está a 1 y las demás señales de carga están a 0, haciendo que el contenido de *R*2 que está en el bus se cargue en *R*0 con el siguiente flanco de subida de reloj. La segunda transferencia de la tabla ilustra la carga del contenido de *R*1 en *R*0 y *R*2. El registro fuente *R*1 se selecciona puesto que *Select* es igual a 01. En este caso, *L*2 y *L*0 están a 1, haciendo que el contenido de *R*1 en el bus se cargue en los registros *R*0 y *R*2. La tercera transferencia es un cambio entre *R*0 y *R*1, es imposible realizarla en un sólo ciclo de reloj, ya que se requieren dos fuentes simultáneas en un sólo bus. Por esto, esta transferencia necesita dos buses al menos o combinar un bus con un camino dedicado de un registro a otro. Véase que tal transferencia se puede ejecutar con los multiplexores dedicados de la Figura 7-19(a). Así, en un sistema con un solo bus, las transferencias simultáneas con diferentes fuentes son imposibles de realizar en un solo ciclo de reloj, mientras que con multiplexores dedicados, cualquier transferencia es posible. Por ello, la reducción de hardware que hay en un solo bus en lugar de multiplexores dedicados da como resultado limitaciones en las transferencias simultáneas.

Si suponemos que solamente se necesitan transferencias que involucran a un sólo registro fuente, entonces podemos usar el circuito de la Figura 7-19 para comparar la complejidad del hardware dedicado frente a sistemas basados en un solo bus. Primeramente, suponemos que el diseño de un multiplexor es como el de la Figura 4-16. En la Figura 7-19(a), hay 2*n* puertas AND y *n* puertas OR por multiplexor (sin contar los inversores), que da un total de 9*n* puertas. Por contra, en la Figura 7-19(b), el bus multiplexor necesita sólo 3*n* puertas AND y *n* puertas OR, que da un total de 4*n* puertas. También, las conexiones de los datos de entrada a los multiplexores se reducen de 6*n* a 3*n*. Así, el coste del hardware de selección se reduce a la mitad.

□ **TABLA 7-13**
Ejemplo de transferencia de registros usando un sólo bus en la Figura 7-19(b)

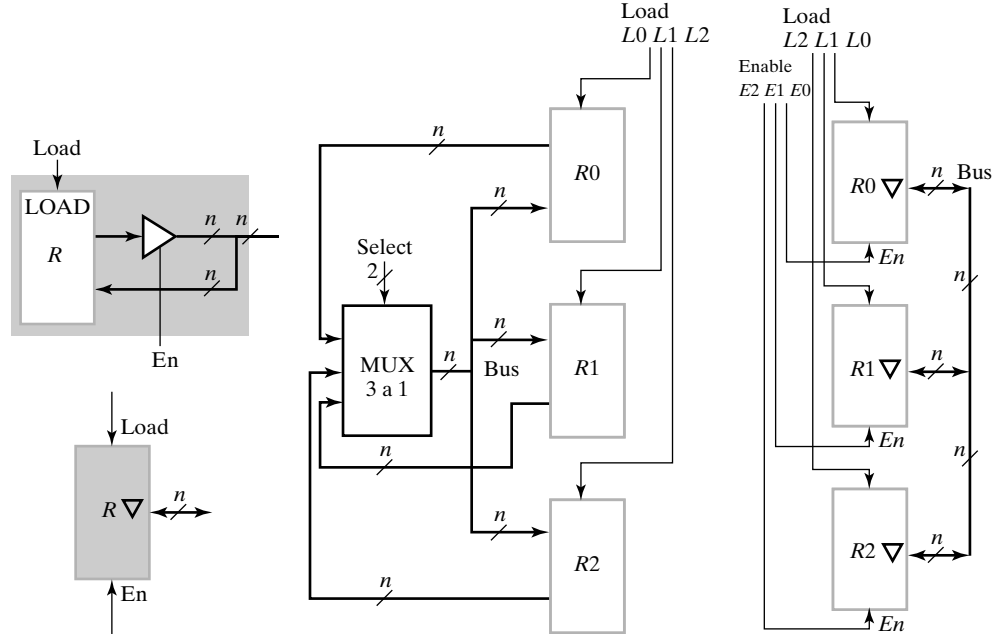
	Selección		**Carga**		
Transferencia de registro	***S*1**	***S*0**	***L*2**	***L*1**	***L*0**
$R0 \leftarrow R2$	1	0	0	0	1
$R0 \leftarrow R1,\ R2 \leftarrow R1$	0	1	1	0	1
$R0 \leftarrow R1,\ R1 \leftarrow R0$	Imposible				

Bus triestado

Un bus se puede construir con buffers triestado (en inglés *three-state buffers*), ya presentados en la Sección 2-8, en lugar de multiplexores. Esto tiene la posibilidad de reducir aún más el número de conexiones. ¿Por qué usar buffers triestado en lugar de multiplexores, en especial, para construir buses? La razón es que las salidas de varios buffers triestado pueden ser conectadas juntas para formar una línea de un bit de un bus, y así el bus se forma sólo con un nivel de puertas lógicas. Por otra parte, en un multiplexor, tal cantidad de fuentes da lugar a puertas OR con fan-in altos, necesitándose múltiples niveles de puertas OR, introduciendo más lógica e incrementando el retardo. Por contra, los buffers triestado proporcionan una forma útil de cons-

truir buses rápidos que, con frecuencia, se prefieren usar en tales casos. Más importante, sin embargo, es el hecho de que las señales pueden viajar en dos direcciones en buses con tercer estado. De esta forma, los buses tri-estado pueden usar la misma interconexión para sacar e introducir datos de un circuito lógico. Esta característica, que es la más importante cuando se cruzan los límites de los chips, se ilustra en la Figura 7-20(a). La figura muestra un registro con *n* líneas que sirven tanto para las entradas como para las salidas que atraviesan el límite del área sombreada. Si los buffers triestado están habilitados, entonces las líneas son de salida; si los buffers están deshabilitados, las líneas pueden ser de entrada. El símbolo para esta estructura se da también en la figura. Véase que las líneas bidireccionales del bus se representan mediante una flecha de doble punta. Además, un pequeño triangulo indica que el registro tiene salidas con tercer estado.

La Figura 7-20(b) y la Figura 7-20(c) muestran la realización de un bus con multiplexor y un bus tri-estado, respectivamente, para su comparación. El símbolo de la Figura 7-20(a) es un registro con entrada-salida bidireccional que se usa en la Figura 7-20(c). En contraste con la situación de la Figura 7-19, donde los multiplexores dedicados se reemplazaron por un bus, estas dos realizaciones son idénticas en términos de su capacidad de transferencia de registros. Véase que, en los buses tri-estado, sólo hay tres conexiones de datos a los registros por cada bit del bus. El bus realizado con el multiplexor tiene seis conexiones de datos por bit del conjunto de los registros. Esta reducción en el número de conexiones de los datos a la mitad, aparte de la su facilidad de construcción, hace del bus tri-estado una atractiva alternativa. El uso de tales líneas bidireccionales es particularmente eficaz entre circuitos lógicos, con líneas de entrada-salida, que están separados físicamente en diferentes encapsulados.



(a) Registro con líneas de entrada-salida bidireccionales y su símbolo

(b) Bus y multiplexor

(c) Bus triestado usando registro con líneas bidireccionales

□ **FIGURA 7-20**
Bus triestado estado versus bus multiplexado

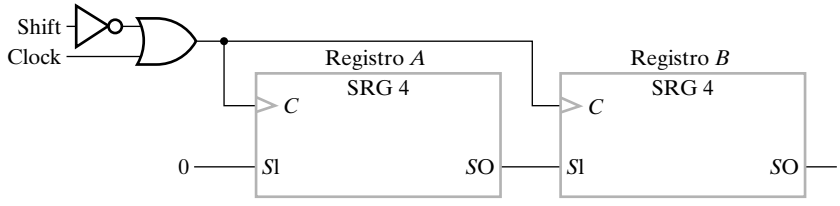
7-9 TRANSFERENCIA SERIE Y MICROOPERACIONES

Un sistema digital se dice que opera en modo serie cuando la información del sistema transfiere o manipula un bit en cada instante de tiempo. La información se transfiere bit a bit desplazando los bits de un registro a otro. Esta forma de transferencia contrasta con la transferencia en paralelo, en la que todos los bits del registro se transfieren simultáneamente.

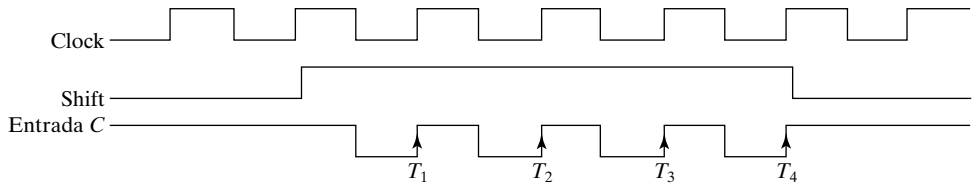
La transferencia serie de información de un registro *A* a otro registro *B* se hace con registros de desplazamiento, como se muestra en la Figura 7-21(a). La salida serie del registro *A* se conecta a la entrada serie del registro *B*. La entrada serie del registro *A* recibe ceros mientras el dato se transfiere al registro *B*. También es posible que el registro *A* reciba otra información binaria, o si queremos mantener el dato en el registro *A*, podemos conectar su salida serie a su entrada serie, de forma que la información vuelva al mismo registro. El contenido inicial del registro *B* se extrae desplazándolo hacia afuera a través de su salida serie y se pierde a no ser que se transfiera al registro *A*, o a un tercer registro u otro tipo de memoria. La entrada de control de desplazamiento, Shift, determina cuándo y cuántas veces se desplaza el registro. Los registros que usan Shift se controlan mediante la lógica de la Figura 7-2, que permite a los pulsos de reloj pasar a la entrada de reloj del registro de desplazamiento cuando Shift vale 1.

En la Figura 7-21, cada registro de desplazamiento tiene cuatro etapas. La lógica que supervisa la transferencia debe diseñarse para habilitar a los registros de desplazamiento, mediante la señal Shift, durante un número fijo de cuatro pulsos de reloj. En la Figura 7-21(b) se muestra el registro de desplazamiento con habilitación mediante lógica aplicada a la entrada de reloj. Los cuatro pulsos encuentran a la señal Shift activada, de forma que a la salida de la lógica conectada a las entradas de reloj, se produce cuatro pulsos: T_1, T_2, T_3 y T_4. Cada transición positiva de estos pulsos produce un desplazamiento en ambos registros. Después del cuarto pulso, Shift regresa a 0 y los registros se deshabilitan. Véase de nuevo que, para cada flanco positivo, los pulsos a la entrada de reloj son 0, y el nivel inactivo cuando no hay pulsos presentes es 1 en lugar de 0.

Supongamos ahora que el contenido del registro *A* antes del desplazamiento es 1011 y el del registro *B* es 0010, y la entrada *SI* del registro *A* esta a 0. Entonces la transferencia serie de *A* a *B* se sucede en cuatro pasos, como se muestra en la Tabla 7-14. Con el primer pulso T_1, el bit



(a) Diagrama de bloques



(b) Cronograma

□ **FIGURA 7-21**
Transferencia serie

□ **TABLA 7-14**
Ejemplo de una transferencia serie

Temporización de los pulsos	Registro de desplazamiento *A*				Registro de desplazamiento *B*			
Valor inicial	1	0	1	1	0	0	1	0
Después de T_1	0	1	0	1	1	0	0	1
Después de T_2	0	0	1	0	1	1	0	0
Después de T_3	0	0	0	1	0	1	1	0
Después de T_4	0	0	0	0	1	0	1	1

más a la derecha de *A* se desplaza al bit más a la izquierda de *B*, el bit más a la izquierda de *A* recibe un 0 por la entrada serie, y en el mismo instante, los bits restantes de *A* y *B* se desplazan una posición a la derecha. Los siguientes tres pulsos llevan a cabo idénticas operaciones, desazando los bits de *A* a *B* uno a uno mientras entran ceros en *A*. Después del cuarto desplazamiento, la lógica que supervisa la transferencia cambia la señal Shift a 0 y los desplazamientos se paran. El registro *B* contiene ahora 1011, que eran los valores anteriores de *A* y el registro *A* contiene ceros.

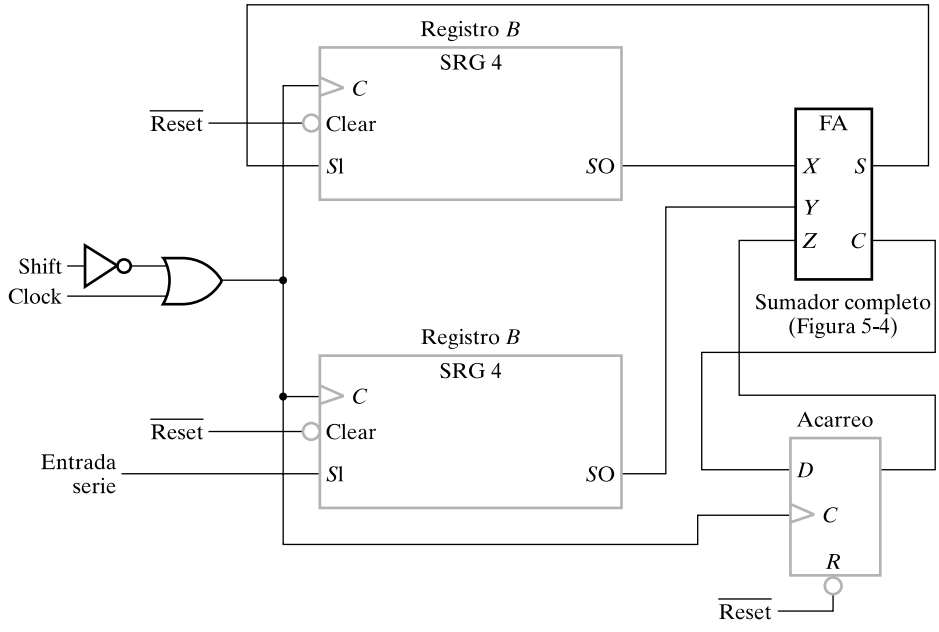
La diferencia entre el modo serie y paralelo debería estar clara después de este ejemplo. En el modo paralelo, la información de todos los bits del registro está disponible y todos los bits se pueden transferir simultáneamente en un ciclo de reloj. En el modo serie, los registros tienen una sola entrada serie y una sola salida serie, y la información se transfiere bit a bit.

Suma en serie

Las operaciones en los sistemas digitales se suelen realizar en paralelo puesto que son más rápidas. Las operaciones en serie son más lentas pero tienen la ventaja de que requieren menos hardware. Para probar el modo de operación serie, mostraremos la forma de operar de un sumador serie. También comparamos el sumador serie con el paralelo que se presentó en la Sección 5-2 para ilustrar la relación espacio-tiempo a la hora de diseñar.

Los dos números binarios a sumar en modo serie deben almacenarse en sendos registros de desplazamiento. Cada pareja de bits se suman en diferente instante de tiempo mediante un sumador completo, FA, (del ingles *full-adder*), como se muestra en la Figura 7-22. El acarreo de salida del sumador completo se transfiere a un flip-flop tipo *D*. La salida de este flip-flop de acarreo se usa como acarreo de entrada para el siguiente par de bits. El bit de suma de la salida S del sumador completo se transfiere a un tercer registro, en nuestro caso se ha elegido para transferir el resultado de la suma el registro *A*, según el contenido se desplaza hacia el exterior. La entrada serie del registro *B* puede, a su vez, recibir un nuevo número binario según su contenido se va desplazando durante la suma.

La forma de operar del sumador serie es como sigue: el registro *A* mantiene un sumando, el registro *B* otro y el flip-flop de acarreo debe haber sido puesto a 0. Las salidas serie de *A* y de *B* proporcionan la pareja de bits a las entradas *X* y en *Y* del sumador completo. La salida del flip-flop de acarreo proporciona el acarreo a la entrada *Z*. Cuando la señal Shift se pone a 1, la puerta OR permite que llegue el reloj a ambos registros y al flip-flop. Cada pulso de reloj desplaza ambos registro a la vez a la derecha, transfiere el bit de suma, *S*, al flip-flop más a la izquierda de *A*, y transfiere el acarreo de salida al flip-flop de acarreo. La lógica de control del desplaza-



□ **FIGURA 7-22**
Suma serie

miento permite que lleguen a los registros tantos pulsos de reloj como bits hay en los registros (cuatro en nuestro ejemplo). En cada pulso se transfiere un nuevo bit de la suma a *A*, un nuevo acarreo al flip-flop y ambos registros se desplazan una vez a la derecha. Este proceso continúa hasta que la lógica que controla los desplazamientos pone a 0 a la señal Shift. De esta forma se lleva a cabo la suma, pasando cada par de bits y el acarreo previo a través del sumador completo y transfiriendo la suma, bit a bit, al registro *A*.

Al principio podemos inicializar los registros *A* y *B* y el flip-flop de acarreo a 0, luego desplazamos el primer número en el registro *B* y posteriormente se suma con el 0 del registro *A*. Mientras se desplaza el primer número de *B* y se va sumando, podemos transferir un segundo número en *B* a través de su entrada serie. El segundo número se puede sumar con el contenido de *A* al mismo tiempo que se transfiere un tercer número al registro *B*. La suma en modo serie se puede repetir para realizar la suma de dos, tres o más números, cuyas sumas se van acumulando en el registro *A*.

La comparación del sumador serie con el sumador paralelo, descrito en la Sección 5-2, nos da un ejemplo de la relación espacio-tiempo. El sumador paralelo tiene *n* sumadores completos, tantos como bits tienen los operandos, mientras que el sumador serie necesita sólo un sumador completo. Excluyendo los registros de ambos, el sumador paralelo es un circuito combinacional mientras que el sumador serie es un circuito secuencial pues incluye el flip-flop de acarreo. El circuito serie necesita *n* ciclos de reloj para completar la suma. Circuitos idénticos, como los *n* sumadores completos del sumador paralelo, se conectan en cadena constituyendo un ejemplo de un *array de células básicas*. Si los valores de los acarreos entre los sumadores completos se consideran como variables de estado, entonces los estados finales, desde el menos significativo al más significativo, son los mismos que los estados que aparecen en la secuencia de salida del sumador serie. Véase que en los arrays de células básicas, los estados aparecen en el espacio (paralelamente), pero en el circuito secuencial aparecen en el tiempo (de forma serie). Pasando

de una forma de realizar el circuito a otra podemos hacer una consideración relativa al espacio-tiempo. El sumador paralelo, en espacio (área) es *n* veces más grande que el sumador serie (ignorando el área de flip-flop de acarreo), pero es *n* veces más rápido. El sumador serie, aunque es *n* veces más lento pero es *n* veces más pequeño. Esto da al diseñador una oportunidad importante de enfatizar su diseño en área o en velocidad, donde más área significa más coste.

7-10 MODELADO EN HDL DE REGISTROS DE DESPLAZAMIENTO Y CONTADORES-VHDL

El ejemplo de un registro de desplazamiento y un contador binario sirven para ilustrar el uso de VHDL para modelar registros y las operaciones sobre su contenido.

EJEMPLO 7-3 Registro de desplazamiento de 4 bits en VHDL

El código en VHDL de la Figura 7-23 modela, a nivel de comportamiento, un registro de desplazamiento. Una entrada `RESET` pone el contenido del registro a cero asíncronamente. El registro de desplazamiento contiene flip-flops y, por tanto, tiene un proceso que describe algo parecido a un flip-flop tipo *D*. Los cuatro flip-flops se representan mediante la señal `shift`, de tipo `std_logic_vector` de tamaño 4. `Q` no se puede usar para representar a los flip-flops puesto que es una salida y las salidas de los flip-flops se usan internamente. El desplazamiento a la izquierda se consigue aplicado el operador concatenación `&` a los tres bits de la derecha de

```
-- Registro de desplazamiento de 4 bits con reset asíncrono

library ieee;
use ieee.std_logic_1164.all;

entity srg_4_r is
  port(CLK, RESET, SI : in std_logic;
    Q : out std_logic_vector(3 downto 0);
    SO : out std_logic);
end srg_4_r;

architecture behavioral of srg_4_r is
signal shift : std_logic_vector(3 downto 0);
begin
process (RESET, CLK)
begin
  if (RESET = '1') then
    shift <= "0000";
  elsif (CLK'event and (CLK = '1')) then
    shift <= shift(2 downto 0) & SI;
  end if;
end process;
  Q <= shift;
  SO <= shift(3);
end behavioral;
```

□ **FIGURA 7-23**
Descripción en VHDL del comportamiento de un registro de desplazamiento a la izquierda de 4 bits

shift y a la entrada de desplazamiento SI. Esta cantidad se transfiere a shift moviendo el contenido de un bit a la izquierda y cargando el valor de SI en el bit más a la derecha. El siguiente proceso que ejecuta el desplazamiento tiene dos sentencias, una que asigna el valor de shift a la salida Q y la otra que define la salida del desplazamiento a la señal SO como el contenido del bit más a la izquierda del desplazamiento. ■

EJEMPLO 7-4 Contador de 4 bits en VHDL

El código VHDL de la Figura 7-24 describe, a nivel de comportamiento, un contador de 4 bits. Una entrada de RESET pone el contenido del contador a cero asíncronamente. El contador tiene flip-flops y, por tanto, tiene un proceso que describe algo parecido a un flip-flop tipo *D*. Los cuatros flip-flops se representan mediante la señal count, de tipo std_logic_vector y de tamaño cuatro. Q no se puede usar para representar a los flip-flops puesto que es una salida y las salidas de los flip-flops se deben usar internamente. La cuenta ascendente se consigue sumando 1, "0001", a count. Como la suma no es una operación para el tipo std_logic_vector, se necesita usar un paquete de la librería ieee, std_logic_unsigned.all, que define operaciones sin signo para el tipo std_logic. Siguiendo el proceso que ejecuta la puesta a 0 y la cuenta, encontramos dos sentencias, una asigna el valor de count a la salida Q, y otra que define la señal de salida de fin de cuenta CO. Se usa una sentencia **when-else** que pone CO a 1 cuando se alcanza el valor máximo de la cuenta cuando EN es igual a 1.

```
-- Contador binario de 4 bits con reset asíncrono

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity count_4_r is
   port(CLK, RESET, EN : in std_logic;
        Q : out std_logic_vector(3 downto 0);
        CO : out std_logic);
end count_4_r;

architecture behavioral of count_4_r is
signal count : std_logic_vector(3 downto 0);
begin
process (RESET, CLK)
begin
   if (RESET = '1') then
      count <= "0000";
   elsif (CLK'event and (CLK = '1') and (EN = '1')) then
       count <= count + "0001";
   end if;
end process;
Q <= count;
CO <= '1' when count = "1111" and EN = '1' else '0';
end behavioral;
```

□ **FIGURA 7-24**
Descripción en VHDL del comportamiento de un contador binario de 4 bits con reset asíncrono ■

7-11 MODELADO EN HDL DE REGISTROS DE DESPLAZAMIENTO Y CONTADORES-VERILOG

El ejemplo de un registro de desplazamiento y un contador binario sirven para ilustrar el uso de Verilog para modelar registros y las operaciones sobre su contenido.

EJEMPLO 7-5 Código Verilog para un registro de desplazamiento

En la Figura 7-25 se describe, a nivel de comportamiento, un registro de desplazamiento a izquierdas. Una entrada `RESET` pone el contenido del registro a cero asíncronamente. El registro de desplazamiento contiene flip-flops, de forma que el proceso de descripción comienza con **`always`** que se asemeja al de un flip-flop de tipo *D*. Los cuatro flip-flops se representan mediante el vector `Q`, de tipo **`reg`** con sus bits numerados de 3 a 0. El desplazamiento a la izquierda se consigue aplicando `{ }` para concatenar los tres bits de la derecha de `Q` y la entrada de desplazamiento `SI`. Esta cantidad se transfiere a `Q` moviendo el contenido de un bit de la izquierda y cargando el valor de `SI` en el bit más a la derecha. Antes del proceso que realiza el desplazamiento hay una sentencia de asignación continua que asigna el contenido del bit más a la izquierda de `Q` a la salida de desplazamiento `SO`.

```
// Registro de desplazamiento de 4 bits con Reset asíncrono

module srg_4_r_v (CLK, RESET, SI, Q,SO);
   input CLK, RESET, SI;
   output [3:0] Q;
   output SO;

reg [3:0] Q;

assign SO = Q[3];

always@(posedge CLK or posedge RESET)
begin
   if (RESET)
      Q <= 4'b0000;
   else
      Q <= {Q[2:0], SI};
end
endmodule
```

□ **FIGURA 7-25**
Descripción Verilog de comportamiento de un registro de desplazamiento a la izquierda de 4 bits con reset asíncrono ■

EJEMPLO 7-6 Código Verilog para un contador

El Código Verilog de la Figura 7-26 modela, a nivel de comportamiento, un contador binario de 4 bits. Una entrada `RESET` pone el contenido del registro a cero asíncronamente. El contador contiene flip-flops y, por tanto, la descripción contiene un proceso que se asemeja a un flip-flop tipo *D*. Los cuatro flip-flops se representan por la señal `Q` de tipo **`reg`** de tamaño cuatro. La

cuenta ascendente se consigue sumando 1 a Q. Antes del proceso que realiza la puesta a cero y la cuenta, hay una sentencia condicional continua que define la salida del fin de cuenta como CO. CO se pone a 1 cuando se alcanza la cuenta máxima y EN es igual a 1. Nótese que la operación lógica AND se hace con el operador &&.

```
// Contador binario de 4 bits con reset asíncrono

module count_4_r_v (CLK, RESET, EN, Q, CO);
   input CLK, RESET, EN;
   output [3:0] Q;
   output CO;

reg [3:0] Q;

assign CO = (count == 4'b1111 && EN == 1'b1) ? 1 : 0;
always@(posedge CLK or posedge RESET)
   begin
   if (RESET)
     Q <= 4'b0000;
   else if (EN)
     Q <= Q + 4'b0001;
   end
endmodule
```

□ **FIGURA 7-26**
Descripción Verilog de comportamiento de un contador binario de 4 bits con reset asíncrono ■

7-12 RESUMEN DEL CAPÍTULO

Los registros son un conjunto de flip-flops, o un conjunto de flip-flops interconectados, y lógica combinacional. El registro más simple es un conjunto de flip-flops que se cargan con los nuevos datos de sus entradas en cada ciclo de reloj. Los más complejos son los registros que pueden cargar nuevos datos bajo una señal de control para unos ciclos de reloj concretos. La transferencia de registros es un medio de representar y especificar operaciones elementales de procesado. La transferencia de registros se puede relacionar con su correspondiente sistema hardware digital, tanto a nivel de bloques como a nivel detallado de la lógica. Las microoperaciones son operaciones elementales que se ejecutan sobre los datos almacenados en registros. Entre las microoperaciones aritméticas se incluyen la suma y la resta, descritas como transferencia de registros y se realizan con su hardware correspondiente. La microoperaciones lógicas, esto es, aplicaciones bit a bit de primitivas lógicas tales como la AND, OR y XOR, combinadas con palabra binarias, que proporcionan una máscara y complementos selectivos en otra palabra binaria. Las microoperaciones de desplazamiento a la izquierda y la derecha mueven los datos lateralmente una o más posiciones sincronizadamente.

Los registros de desplazamiento aportan una nueva dimensión a la transferencia de datos, ya que están diseñadas para mover la información lateralmente bit a bit en cada instante de tiempo. Si se combinan con la posibilidad de cargar datos, se pueden usar para convertir datos en formato paralelo a formato serie. Así mismo, si las salidas del registro están accesibles, un registro de desplazamiento se puede usar para convertir datos en formato serie a formato paralelo. Este movimiento lateral de datos puede usarse también en estructuras hardware que ejecutan operaciones en serie.

Los contadores se usan para conseguir una secuencia determinada de valores, normalmente como una cuenta ordenada en binario. El contador más simple no tiene más entradas que la de reset asíncrono para su inicialización a cero. Este tipo de contadores simplemente cuentan pulsos de reloj. Versiones más complejas admiten la carga de datos y tienen señales que lo habilitan para contar.

Los multiplexores seleccionan entre múltiples caminos de transferencia que entran en un registro. Los buses son caminos para transferencias de registros que comparten caminos y ofrecen la posibilidad de reducir hardware a cambio de limitaciones en casos de transferencias simultáneas. Además de los multiplexores, los buffers tri-estado proporcionan caminos para transferencias bidireccionales y reducen el número de conexiones.

REFERENCIAS

1. MANO, M. M.: *Digital Design*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

2. WAKERLY, J. F.: *Digital Design*: *Principles and Practices*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2000.

3. *IEEE Standard VHDL Language Reference Manual*. (ANSI/IEEE Std 1076-1993; revision of IEEE Std 1076-1987). New York: The Institute of Electrical and Electronics Engineers, 1994.

4. *IEEE Standard Description Language Based on the Verilog*(*TM*) *Hardware Description Language* (IEEE Std 1364-1995). New York: The Institute of Electrical and Electronics Engineers, 1995.

5. THOMAS, D. E., AND P. R. MOORBY: *The Verilog Hardware Description Language* 4th ed. Boston: Kluwer Academic Publishers, 1998.

PROBLEMAS

El signo (+) indica problemas más avanzados y el asterisco (*) indica que hay una solución disponible en la dirección de Internet: http://www.librosite.net/Mano.

7-1. Mediante simulación manual o por computadora, demuestre que la función de habilitación de reloj de la Figura 7-1(c) funciona correctamente con el registro de la Figura 7-1(a). Use un flip-flop disparado con flanco de subida con *Clock* como entrada de reloj para generar *Load*. Asegúrese de que usa puertas y flip-flops sin retardos.

7-2. +Cambie la puerta OR de la Figura 7-1(c) por una puerta AND y quite el inversor colocado sobre la señal Load.

(a) Haga la misma simulación que en el Problema 7-1 para demostrar que el nuevo circuito no funciona correctamente. Explique porqué.

(b) ¿Funcionará correctamente el circuito si el flip-flop que genera Load se dispara con flanco de bajada en lugar del flanco de subida de Clock?

7-3. Suponga que los registros $R1$ y $R2$ de la Figura 7-6 contienen dos números sin signo. Cuando la entrada de selección, X, es igual a 1, el circuito sumador-restador ejecuta la operación aritmética «$R1$ + complemento a 2 de $R2$.» Esta suma y el acarreo de salida C_n se transfiere a $R1$ y C cuando $K_1 = 1$ y, además, se produce un flanco de subida de reloj.

(a) Demuestre que si $C = 1$, entonces el valor transferido a $R1$ es igual a $R1 - R2$, pero si $C = 0$, el valor transferido a $R1$ es el complemento a 2 de $R2 - R1$.

(b) Indique cómo se puede usar el valor del bit *C* para detectar un acarreo (*borrow*) después de la resta de dos números sin signo.

7-4. *Realice las lógica bit a bit (*bitwise*) AND, OR y XOR de los dos operandos de 8 bits 10011001 y 11000011.

7-5. Dado el operando de 16 bits 00001111 10101010 ¿qué operación se debe realizar y qué operando se debe usar

(a) Para poner a cero todos los bits en posiciones pares? (Suponga que la posición de los bits van de 15 a 0 desde la izquierda a la derecha).
(b) Para cambiar a 1 los 4 bits más a la izquierda.
(c) Para complementar los 8 bits centrales.

7-6. *Partiendo de un operando de 8 bits 01010011, obtenga los valores después de cada microoperación de desplazamiento dadas en la Tabla 7-5.

7-7. *Modifique el registro de la Figura 7-11 de forma que opere de acuerdo con la siguiente tabla de funcionamiento usando las entradas de modo de selección S_1 y S_0.

S_1	S_0	Operación del registro
0	0	No cambia
0	1	Pone el registro a 0
1	0	Desplazamiento hacia abajo
1	1	Carga paralelo de datos

7-8. *Un contador en anillo es un registro, como el de la Figura 7-9, con la salida serie conectada a la entrada serie.

(a) Partiendo de un estado inicial 1000, indique la secuencia de cada estado de los cuatro flip-flops en cada desplazamiento.
(b) Comenzando en el estado 10.0 ¿cuántos estados hay en la secuencia de un contador en anillo de *n* bits?

7-9. Un Contador Johnson usa la salida serie complementada de un registro de desplazamiento como entrada serie.

(a) Partiendo del estado inicial 0000, indique la secuencia de estados después de cada desplazamiento hasta que el registro alcanza de nuevo el estado 0000.
(b) Comenzando por el estado 00.0 ¿Cuántos estados hay en la secuencia de *n* bit del Contador Johnson?

7-10. ¿Cuántos valores de los flip-flops de un contador asíncrono de 8 bits se complementan al alcanzar el valor de cuenta siguiente a

(a) 11101111? **(b)** 01111111?

7-11. +Para la familia lógica CMOS, el consumo de potencia es proporcional al número total de transiciones de 1 a 0 y 0 a 1 de todas las entradas y salidas de las puertas del circuito. Cuando se diseña un contador de muy bajo consumo, los contadores asíncronos se prefieren frente a los síncronos. Cuente cuidadosamente el número de cambios en las salidas y las entradas, incluidas las debidas al reloj, para un ciclo completo de valores de un contador asíncrono de 4 bits frente a un contador síncrono de la misma longitud. Basándose en

este examen, explique por qué el contador asíncrono es superior en términos de consumo de potencia.

7-12. Construya un contador serie-paralelo de 16 bits, usando cuatro contadores de 4 bits. Suponga que toda la lógica empleada son puertas AND y que los cuatro contadores se conectan en serie. ¿Cuál es el número máximo de puertas encadenadas por las que una señal se debe propagar a través del contador de 16 bits?

7-13. + Se desea diseñar un contador síncrono paralelo de 64 bits.

(a) Dibuje el diagrama lógico de un contador paralelo de 64 bits usando bloques contadores de 8 bits y 2 niveles de conexiones de puertas en paralelo entre los bloques. En estos bloques, *CO* no depende de *EN*.

(b) ¿Cuál es la relación entre la frecuencia de este contador y la del contador serie-paralelo de 64 bits? Suponga que el tiempo de propagación de un flip-flop tipo *D* es el doble que el de una puerta AND y que el tiempo de set-up es igual al retardo de una puerta AND.

7-14. Uniendo el contador binario de la Figura 7-13 y una puerta AND, construya un contador que cuente de 0000 hasta 1010. Repítalo para una cuenta de 0000 hasta 1010. Repítalo para una cuenta de 0000 a 1110. Minimice el número de entradas de la puerta AND.

7-15. Usando dos contadores binarios del tipo mostrado en la Figura 7-13 y puertas lógicas, construya un contador binario que cuente desde 9 hasta 129 en decimal. Añada una entrada al contador que inicialice síncronamente a 9 cuando la señal INIT es 1.

7-16. *Verifique las ecuaciones de entrada de los flip-flops de un contador síncrono BCD especificadas en la Tabla 7-9. Dibuje el diagrama lógico de un contador BCD con entrada de habilitación.

7-17. *Use flip-flops tipo *D* y puertas para diseñar un contador binario con la siguiente secuencia:

(a) 0, 1, 2 **(b)** 0, 1, 2, 3, 4, 5

7-18. Utilice flip-flops tipo *D* y puertas para diseñar un contador con la siguiente secuencia binaria: 0, 1, 3, 2, 4, 6.

7-19. Use solamente flip-flops tipo *D* para diseñar un contador que repita la secuencia binaria: 0, 1, 2, 4, 8.

7-20. Dibuje el diagrama lógico de un registro de 4 bits con entrada de modo de selección S_1 y S_0. El registro debe operar de acuerdo con la siguiente tabla de funcionamiento:

S_1	S_0	Operación del registro
0	0	No cambia
0	1	Pone el registro a 0
1	0	Desplazamiento hacia abajo
1	1	Carga paralelo de datos

7-21. *Muestre el diagrama hardware que permite la siguiente sentencia de transferencia de registros:

$$C_3\colon R2 \leftarrow R1,\ R1 \leftarrow R2$$

7-22. Las salidas de los registros *R*0, *R*1, *R*2 y *R*3 se conectan mediante un multiplexor de 4 a 1 a las entradas del registro *R*4. Cada registro es de 8 bits. Las transferencias pedidas, según indican las variables de control, son:

$$C_0\text{: } R4 \leftarrow R0$$
$$C_1\text{: } R4 \leftarrow R1$$
$$C_2\text{: } R4 \leftarrow R2$$
$$C_3\text{: } R4 \leftarrow R3$$

Las variables de control se excluyen mutuamente (es decir, sólo una variable puede ser 1 en cualquier instante de tiempo) mientras que las otras tres restantes son iguales a 0. Además, no se realiza ninguna transferencia si todas las señales de control son iguales a 0.

(a) Usando registros y multiplexores, dibuje un diagrama lógico detallado del hardware que permite la transferencia de un solo bit entre estos registros.

(b) Dibuje el diagrama lógico de una lógica sencilla para que, a partir de las variables de control, se gobiernen las dos variables del multiplexor y las señales de carga de los registros.

7-23. *Usando dos registros de 4 bits, *R*1 y *R*2, puertas AND, OR e inversores, dibuje un diagrama lógico para un solo bit que permita ejecutar las siguientes sentencias:

$C_0\text{: } R2 \leftarrow 0$	Pone a 0 síncronamente a *R*2
$C_1\text{: } R2 \leftarrow \overline{R2}$	Complemente *R*2
$C_2\text{: } R2 \leftarrow R1$	Transfiera *R*1 a *R*2

Las variables de control se excluyen mutuamente (es decir, sólo una variable puede ser 1 en cualquier instante de tiempo) mientras que las otras dos restantes son iguales a 0. Además, no se realiza ninguna transferencia a *R*2 si todas las variables de control son iguales a 0.

7-24. Se quiere diseñar una célula básica de un registro para formar un registro de 8 bits, *A*, que permita realizar las siguientes funciones de transferencia de registros:

$$C_0\text{: } A \leftarrow A \wedge B$$
$$C_1\text{: } A \leftarrow A \vee \overline{B}$$

7-25. Se quiere diseñar una célula básica de un registro para formar un registro de 8 bits, *R*0, que permita realizar las siguientes funciones de transferencia de registros:

$$\overline{S_1} \cdot \overline{S_0}\text{: } R0 \leftarrow 0$$
$$\overline{S_1} \cdot \overline{S_0}\text{: } R0 \leftarrow R0 \vee R1$$
$$S_1 \cdot \overline{S_0}\text{: } R0 \leftarrow R0 \oplus R1$$
$$S_1 \cdot S_0\text{: } R0 \leftarrow R0 \wedge R1$$

Encuentre la mejor lógica usando puertas AND, OR e inversores para las entradas *D* del flip-flop tipo *D* de la célula.

7-26. Se quiere diseñar una célula básica de un registro *B*, que permita realizar las siguientes transferencias de registros:

$$S_1: B \leftarrow B + A$$
$$S_0: B \leftarrow B + 1$$

Comparta la lógica combinacional entre los dos registros tanto como le sea posible.

7-27. Se diseña una lógica para realizar transferencias entre tres registros *R*0, *R*1 y *R*2. Las variables de control son las dadas en el Problema 7-2. Las transferencias entre registros son las siguientes:

$$C_A: R1 \leftarrow R0$$
$$C_B: R0 \leftarrow R1,\ R2 \leftarrow R0$$
$$C_C: R1 \leftarrow R2,\ R0 \leftarrow R2$$

Empleando registros y multiplexores dedicados, dibuje el diagrama lógico detallado del hardware, para un bit, que permita estas transferencias de registros.

Dibuje el diagrama lógico que convierte las variables de control C_A, C_B y C_C en entradas de selección de los multiplexores, SELECT, y señales de carga, LOAD, para los registros.

7-28. *Se dan dos sentencias de transferencia de registros (de lo contrario, *R*1 permanece sin cambiar):

$C_1: R1 \leftarrow R1 + R2$ Suma *R*2 a *R*1

$\bar{C}_1 C_2: R1 \leftarrow R1 + 1$ Incrementa *R*1

(a) Usando un contador de 4 bits con carga en paralelo, como el de la Figura 7-1, y un sumador de 4 bits, como el de la Figura 5-5, dibuje el diagrama lógico que ejecuta estas transferencias de registros.

(b) Repita el apartado (a) usando un sumador de 4 bits como el de la Figura 5-5 más las puertas que considere necesarias. Compárelo con lo obtenido en el apartado (a).

7-29. Repita el Problema 7-27 utilizando un bus basado en multiplexores y una conexión directa de un registro a otro en lugar de multiplexores dedicados.

7-30. Dibuje el diagrama lógico de un circuito similar al que se muestra en la Figura 7-7 pero usando buffers tri-estado y un decodificador en lugar de multiplexores.

7-31. *Un sistema tiene el siguiente conjunto de transferencia de registros y se diseña empleando buses:

$$C_a: R0 \leftarrow R1$$
$$C_b: R3 \leftarrow R1,\ R1 \leftarrow R4,\ R4 \leftarrow R0$$
$$C_c: R2 \leftarrow R3,\ R0 \leftarrow R2$$
$$C_d: R2 \leftarrow R4,\ R4 \leftarrow R2$$

(a) Para cada registro destino, enumere todos los registros fuentes.

(b) Para cada registro fuente, enumere todos los registros destino.

(c) Considerando que cada una de las transferencias debe ocurrir simultáneamente ¿cuál es el número mínimo de buses que se deben usar para realizar el conjunto de transferencias dadas? Suponga que a cada registro le llega un solo bus a su entrada.

(d) Dibuje un diagrama de bloques del sistema que muestre los registros y buses y las conexiones entre ellos.

7-32. Las siguientes transferencias de registro deben ejecutarse en un máximo de dos ciclos de reloj:

$$R0 \leftarrow R1 \qquad R8 \leftarrow R3$$
$$R2 \leftarrow R1 \qquad R9 \leftarrow R4$$
$$R4 \leftarrow R2 \qquad R10 \leftarrow R4$$
$$R6 \leftarrow R3 \qquad R11 \leftarrow R1$$

(a) ¿Cuál es el número mínimo de buses necesarios? Suponga que a la entrada de un registro sólo puede llegar un bus y que cualquier red conectada a un registro se contabiliza como un bus.

(b) Dibuje un diagrama lógico que conecte registros y multiplexores para realizar dichas transferencias.

7-33. ¿Cuál es el mínimo número de ciclos de reloj necesarios para realizar el siguiente conjunto de transferencia de registros usando 2 buses?

$$R0 \leftarrow R1 \qquad R7 \leftarrow R1$$
$$R2 \leftarrow R3 \qquad R8 \leftarrow R4$$
$$R5 \leftarrow R6 \qquad R9 \leftarrow R3$$

Suponga que sólo un bus puede conectarse a la entrada de un registro y que cualquier red conectada a un registro se contabiliza como un bus.

7-34. *El contenido de un registro de 4 bits es 0000 inicialmente. El registro se desplaza 8 veces a la derecha con la siguiente secuencia en la entrada serie. El bit más a la izquierda es el que entra primero. ¿Cuál es el contenido del registro después de cada desplazamiento?

7-35. El sumador serie de la Figura 7-22 utiliza 2 registros de 4 bits. El registro *A* contiene el número binario 0111 y el registro *B* contiene 0101. El flip-flop de acarreo está inicialmente a 0. Enumere los valores del registro *A* y del flip-flop de acarreo para cada cuatro desplazamientos.

Todos los ficheros a los que se hace referencia en los siguientes problemas están disponibles en formato ASCII para su simulación y edición en la siguiente dirección de Internet: http://www.librosite.net/Mano. Es necesario disponer de un compilador/simulador de VHDL o Verilog para la simulación de los problemas o parte de ellos. Los modelos se pueden describir, pero con dificultad, si no se dispone de estas herramientas de compilación o simulación.

7-36. *Escriba una descripción de comportamiento en VHDL del registro de 4 bits de la Figura 7-1(a). Compile y simule su descripción para comprobar su corrección.

7-37. Repita el Problema 7-36 para modelar el registro de 4 bits con carga paralelo de la Figura 7-2.

7-38. Escriba una descripción en VHDL para el contador binario de 4 bits de la Figura 7-13 usando un registro con flip-flops tipo *D* y las ecuaciones booleanas de la lógica. Compile y simule su descripción para comprobar su corrección.

7-39. *Escriba una descripción de comportamiento en Verilog del contador de 4 bits de la Figura 7-1(a). Compile y simule su descripción para comprobar su corrección.

7-40. Repita el Problema 5-39 del registro de 4 bits con carga paralelo de la Figura 7-2.

7-41. Escriba una descripción en Verilog del contador binario de 4 bits de la Figura 7-13 usando registros de 4 bits y las ecuaciones booleanas de la lógica. Compile y simule su descripción para comprobar su corrección.

CAPÍTULO

8

SECUENCIAMIENTO Y CONTROL

En el Capítulo 7 introducimos el concepto de ruta de datos para el procesado de datos y el diseño de rutas de datos usando registros y transferencias de registros. En este capítulo nos centraremos en la unidad de control, que también se presentó en el Capítulo 7. Los sistemas digitales se pueden clasificar en programables o no programables dependiendo del tipo de unidad de control. Un sistema no programable tiene entradas pero no tiene un mecanismo para la ejecución de programas. Los sistemas programables son capaces de ejecutar programas. En este capítulo nos centraremos en los sistemas no programados usando como ejemplo un multiplicador. El estudio de los sistemas programables comenzará en el Capítulo 10.

El algoritmo de máquina de estados (en inglés *the algorithmic state machine* (ASM)) es una versión más amigable de los diagramas de estado de los circuitos secuenciales que proporciona una representación del comportamiento de la unidad de control, así como de las transferencias controladas entre registros. Usando las transferencias de registros en ASM se puede representar el comportamiento combinado de una unidad de control y una ruta de datos.

En este capítulo, la unidad de control representada mediante ASM se lleva a cabo usando control cableado. Entre las técnicas empleadas para diseño de control cableado, vamos a considerar dos métodos que simplifican el diseño de unidades de control grandes, si los comparamos con el método de diseño de circuitos secuenciales básicos vistos en el Capítulo 6. El diseño de unidades de control usando control microprogramado se tocará muy brevemente.

Los principales temas de este capítulo son los algoritmos de máquinas de estado, control cableado y representación de algoritmos de máquinas de estados usando HDLs. Como estas técnicas de diseño son bastante generales, van a tener impacto en la mayoría de las partes de la computadora genérica presentada a principios del Capítulo 1. Puesto que la CPU y la FPU del procesador contienen importantes controles para la activación y secuenciación de las operaciones de transferencia de registros, el material que se presenta en este capítulo se va a aplicar en el diseño de dicho procesador

8-1 LA UNIDAD DE CONTROL

La información almacenada en un procesador digital se puede clasificar en datos y en información de control. Como vimos en anteriores capítulos, los datos se manipulan en una ruta de datos usando microoperaciones llevadas a cabo mediante transferencias de registros. Estas operaciones se realizan con sumadores-restador, registros de desplazamiento, multiplexores y buses. La unidad de control proporciona señales que activan las microoperaciones de la ruta de datos para llevar a cabo tareas concretas. La unidad de control también determina la secuencia en la que se realizan las diversas operaciones. El diseño de un sistema digital se trata, en general, dividiéndolo en dos partes bien diferenciadas, los registros y sus transferencias, tema cubierto en el Capítulo 7, y la unidad de control, que se trata en este capítulo.

Generalmente, la temporización de todos los registros en un sistema digital síncrono se controla mediante un reloj maestro. Los pulsos de reloj se aplican a todos los flip-flops y registros del sistema, incluyendo los de la unidad de control. Para evitar que cambie el contenido de los registros en cada pulso de reloj, algunos de ellos llevan una señal de control de carga que habilita o deshabilita la carga de un nuevo dato en dichos registros. Las variables binarias que controlan las entradas de selección de los multiplexores, los buses, la lógica de procesamiento y las entradas de control de carga de los registros se generan, por lo general, en la unidad de control.

La unidad de control, que genera las señales para la secuenciación de las microoperaciones, es un circuito secuencial cuyos estados gobiernan las señales de control del sistema. En cualquier instante, el estado actual de un circuito secuencial activa un determinado conjunto de microoperaciones ya previsto de antemano. Usando las condiciones de estado y las entradas de control, la unidad de control determina su próximo estado. El circuito digital que actúa como controlador proporciona una secuencia de señales para la activación de las microoperaciones y, a su vez, determina su siguiente estado.

En el diseño global de un sistema, podemos distinguir dos tipos distintos de unidades de control en los sistemas digitales: para sistemas programables y para sistemas no programables.

En un *sistema programable*, parte de la entrada al procesador está formada por una secuencia de *instrucciones*. Cada instrucción especifica la operación que el sistema va a ejecutar, qué operandos usar, dónde colocar los resultados de la operación y, en algunos casos, cuál va a ser la siguiente instrucción que se va a ejecutar. En los sistemas programables se suelen almacenar las instrucciones en una memoria, RAM o ROM. Para ejecutar una secuencia de instrucciones es necesario proporcionar la dirección de memoria que va a ser ejecutada. Esta dirección parte de un registro llamado contador de programa, *PC*, (del inglés *program counter*). Como su propio nombre indica, el *PC* posee una lógica que permite su cuenta. Además, para cambiar la secuencia de operaciones, basándose en la información del estado de la ruta de datos, el *PC* necesita tener la posibilidad de realizar una carga en paralelo. Así, en el caso de un sistema programable, la unidad de control contiene un *PC* con lógica de control así como lógica para interpretar la instrucción. Ejecutar una instrucción significa la activación de la secuencia de microoperaciones necesarias en una determinada ruta de datos para desarrollar la operación especificada por la instrucción.

En un *sistema no programable*, la unidad de control no es responsable de obtener instrucciones de la memoria ni es responsable de la secuencia de ejecución de estas instrucciones. No hay *PC* ni registro similar en un sistema así. En cambio, la unidad de control determina las operaciones a ejecutar y la secuencia de dichas operaciones, basándose en la información de sus entradas y los bits de estado de la ruta de datos.

Este capítulo se centra en el diseño de sistemas no programables. Para su ilustración se emplea el algoritmo de máquinas de estados (ASM) para el diseño de la unidad de control, además de técnicas especiales para diseño de ASMs. Los sistemas programables se tratan en los Capítulos 10 y 12.

8-2 ALGORITMO DE MÁQUINAS DE ESTADOS

Una tarea de procesamiento se puede definir mediante unas microoperaciones de transferencia de registros controladas por un mecanismo de secuenciación. Tal tarea se puede especificar como un algoritmo hardware consistente en un número finito de procedimientos que realizan dicha tarea de procesamiento. La parte más estimulante y creativa del diseño digital es el planteamiento de algoritmos hardware que lleven a conseguir los objetivos requeridos. Un algoritmo hardware se puede usar como base para diseñar la ruta de datos y la unidad de control de un sistema.

Una forma recomendable de especificar los pasos de un procedimiento y los caminos de decisión de un algoritmo es un diagrama de flujo. Un diagrama de flujo para un algoritmo hardware debe tener unas características especiales que liguen de cerca el desarrollo hardware de un algoritmo determinado. Para ello usamos un diagrama de flujo llamado *algoritmo de máquinas de estado* (ASM) para definir algoritmos para hardware digital. Por tanto, una máquina de estados es otro término para designar a un circuito secuencial.

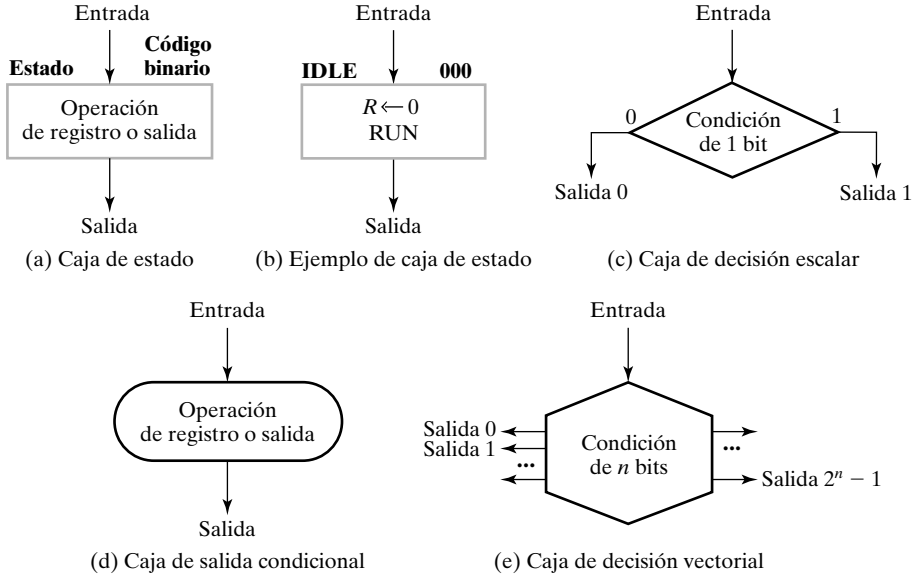
El diagrama ASM se asemeja a un diagrama de flujo convencional pero se interpreta de diferente forma. Un diagrama de flujo convencional describe los pasos del procedimiento sin establecer una relación temporal. Por contra, un diagrama ASM proporciona, no sólo la secuencia de eventos, sino que además describe la relación temporal entre estados de la unidad de control y la acciones de la ruta de datos que ocurren en los estados como respuesta a los pulsos de reloj.

Diagrama ASM

El diagrama ASM contiene tres elementos básicos: la caja de estado, la caja de decisión escalar y la caja de salida condicional, como se ilustra en la Figura 8-1. Por conveniencia, se ha añadido un cuarto elemento, la caja de decisión vectorial. Este componente adicional simplifica la representación de caminos de decisión múltiples y establece una correspondencia entre las representaciones HDLs y los diagramas ASM.

Un estado en la secuencia de control se indica mediante una *caja de estado*, como se muestra en la Figura 8-1(a). La caja de estado es un rectángulo que contiene una operación de transferencia de registro o señales de salida que se activan cuando la unidad de control está en un determinado estado. Implícitamente, la activación de una señal de salida indica la asignación de un 1 a dicha señal. El nombre simbólico del estado se coloca en la esquina superior izquierda de la caja y el código binario del estado, si se le ha asignado, se coloca en la esquina superior derecha de la caja.

La Figura 8-1(b) muestra un ejemplo concreto de una caja de estado. El estado tiene el nombre simbólico IDLE y el código binario asignado es el 000. Dentro de la caja está la transferencia de registro $R \leftarrow 0$ y la salida *RUN*. La transferencia de registro indica la puesta a 0 del registro *R* en cualquier pulso de reloj que ocurra cuando el control está en el estado IDLE. *RUN* indica que la señal de salida es 1 durante el tiempo en que el control esté en el estado IDLE. *RUN* es 1 para cualquier caja de estado en la que aparezca y es 0 para cualquier caja de estado en la que no aparezca.



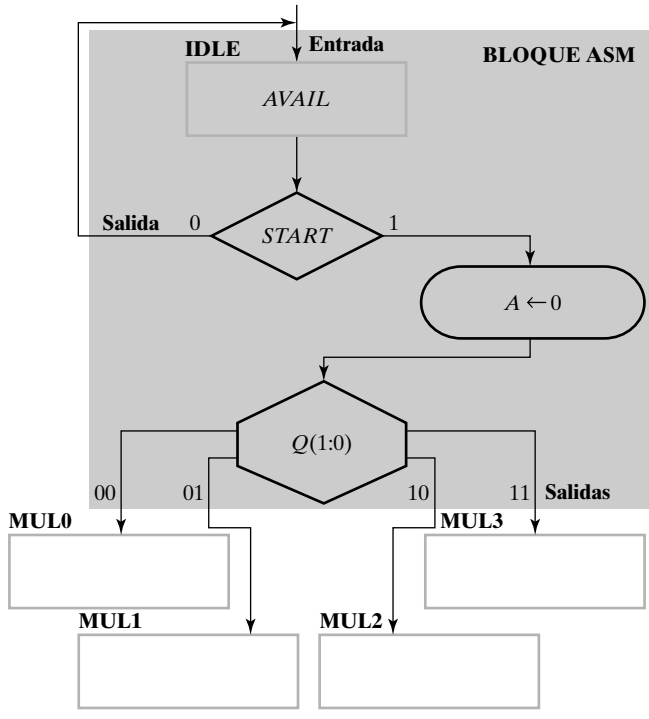
□ **FIGURA 8-1**
Elementos del diagrama ASM

La *caja de decisión escalar* describe el efecto de una entrada en el control. Tiene la forma de un rombo con dos caminos de salida, como se muestra en la Figura 8-1(c). La condición de entrada es una variable binaria de entrada o una expresión booleana que depende solamente de las entradas. Un camino de los dos existentes se toma si la condición de entrada es verdadera (1) y el otro se toma si la condición de entrada es falsa (0).

El tercer elemento, la *caja de salida condicional* es exclusiva del diagrama ASM. La forma ovalada de la caja se muestra en la Figura 8-1(d). Las esquinas redondeadas la diferencian de la caja de estados. El camino de entrada a una caja de salida condicional, procedente de una caja de estado, debe pasar a través de una o más cajas de decisión. Si la condición especificada en el camino, a través de las cajas de decisión que conducen de una caja de estado a una caja de salida condicional, se cumple, se activan las transferencias de registros o salidas enumeradas dentro de la caja de salida condicional.

La *caja de decisión vectorial*, mostrada en la Figura 8-1(e), describe el efecto de un vector de entradas en el control. Su forma hexagonal tiene un máximo de 2^n caminos posibles para un vector binario de n elementos. La entrada condicional es un vector de $n > 1$ variables binarias de entrada o expresiones booleanas que dependen sólo de las entradas. Si el valor del vector coincide con la etiqueta correspondiente a uno de los caminos existentes, se selecciona dicho camino.

Un bloque ASM consiste en una caja de estados y todas las cajas de decisión y de salida condicionales conectadas entre la salida de la caja de estado y los caminos de entrada a la misma caja de estado u otra caja de estado. Un ejemplo de un bloque ASM se muestra en la Figura 8-2. El bloque representa decisiones y acciones de salida que pueden llevarse a cabo en dicho estado. Cualquier salida, para la que se satisfacen las condiciones del bloque ASM, se activa en dicho bloque. Cualquier transferencia de registro para la que se satisfacen las condiciones dentro del bloque ASM será ejecutada cuando ocurra un evento de reloj. Este mismo evento de reloj transferirá el control al siguiente estado, como se especifica en las decisiones pertenecientes al



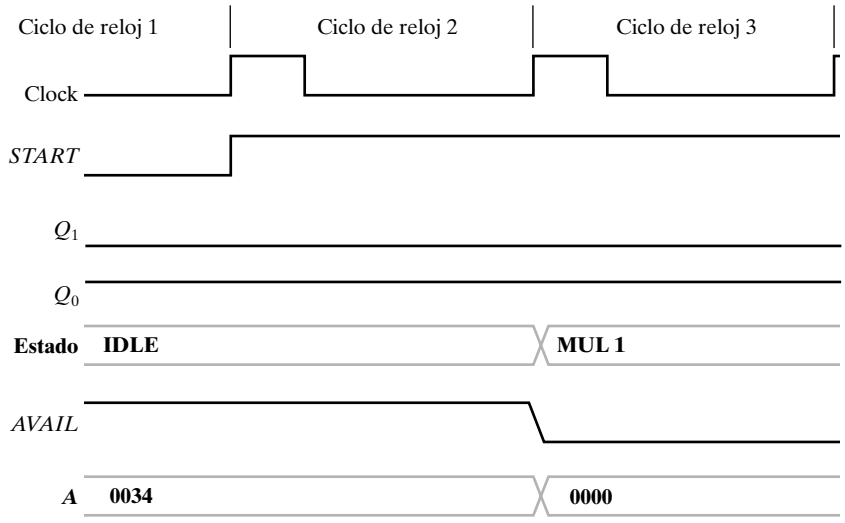
□ **FIGURA 8-2**
Bloque ASM

bloque ASM. El estado del bloque de la Figura 8-2 es IDLE. Durante el estado IDLE la salida AVAIL es 1. Si START es 1 con el flanco activo de reloj, *A* se pone a 0 y, dependiendo de los valores del vector *Q*(1:0), el siguiente estado será MUL0, MUL1, MUL2 o MUL3. En la figura, el camino de entrada y los cinco caminos de salida del bloque ASM se etiquetan en el contorno de dicho bloque.

El diagrama ASM es una forma real de diagrama de estados para parte del circuito secuencial de la unidad de control. Cada caja de estados es equivalente a un nodo del diagrama de estados. Las cajas de decisión son equivalentes a los valores de entrada en las líneas que conectan a los nodos del diagrama. La transferencia de registros, las salidas de las cajas de estado y las cajas de salida condicionales se corresponden a la salida de un circuito secuencial. Las salidas de una caja de estado son las que deberían estar especificadas para el estado de un diagrama de estados de una máquina de De Moore. Las salidas de una caja de salida condicional se corresponden con los valores de las entradas en las líneas que conectan a los estados en un diagrama de estados. Esta dependencia de las entradas se correspondería con una Máquina de estados de Mealy. Si todas las dependencias en un ASM se corresponde con las dependencias de tipo Moore (esto es, si no hay cajas de salidas condicionales), el ASM es una máquina de De Moore. Si hay una o más cajas condicionales con dependencia tipo Mealy, el ASM es de tipo Mealy.

Consideraciones temporales

Para clarificar las consideraciones temporales de un ASM, usamos el ejemplo del bloque ASM de la Figura 8-2. En la Figura 8-3 se ilustra la temporización de los eventos relacionados con el



□ **FIGURA 8-3**
Comportamiento temporal en un diagrama ASM

estado IDLE. Para considerar esta temporización de eventos, resaltamos que sólo usamos flip-flops disparados con el flanco de subida. Durante el primer ciclo de reloj, ciclo 1, la unidad de control está en el estado IDLE y la salida AVAIL es 1 y la entrada START es 0. Basándonos en el bloque ASM, cuando llega el flanco de subida del reloj, el estado permanece en IDLE y AVAIL sigue siendo 1. Además, el contenido del registro *A* permanece sin cambiar. En el ciclo 2, START es 1. Así, cuando llega el flanco de reloj, el registro *A* se pone a 0. Con START a 1, se evalúa *Q*(1:0) y se encuentra que es 01. Para este valor, cuando ocurre el flanco de reloj, el siguiente estado pasa a ser MUL1. El nuevo estado MUL1 y el nuevo valor de *A* aparecen al comienzo de ciclo de reloj 3. El valor de AVAIL pasa a ser 0 puesto que AVAIL no aparece en la caja de estado de MUL1. Véase que la salida AVAIL = 1 aparece concurrentemente con la presencia del estado IDLE pero el resultado de la transferencia del registro *A* aparece concurrentemente con el siguiente estado MUL1. Esto es así porque las salidas cambian asíncronamente en respuesta al estado y al valor de las entradas pero, tanto las transferencias de registro y los cambios de estados, esperan al siguiente flanco de subida del reloj.

8-3 EJEMPLOS DE DIAGRAMAS ASM

Se usa un multiplicador binario para ilustrar una propuesta de diagrama ASM. El multiplicador multiplica 2 números enteros de *n* bits sin signo y dan como resultado un entero de 2*n* bits.

Multiplicador binario

En este ejemplo presentamos un algoritmo hardware para la multiplicación binaria, proponiendo una sencilla ruta de datos y una unidad de control y, posteriormente, describimos sus transferencias de registros y su control mediante el uso de un diagrama ASM. El sistema que se ilustra multiplica dos números binarios sin signo. En la Sección 5-5 se presentó un algoritmo hardware para realizar la multiplicación de forma combinacional, sin elementos de memoria, que utilizaba varios sumadores y puertas AND. Por contra, el algoritmo hardware que se describe ahora, es

un circuito combinacional que sólo usa un sumador y un registro de desplazamiento. Aquí se ilustra el algoritmo, se propone una estructura de transferencia de registros y se formula un diagrama ASM.

ALGORITMO DE LA MULTIPLICACIÓN La multiplicación de dos números binarios sin signo, realizada a mano en papel, se lleva a cabo mediante desplazamientos sucesivos del multiplicando a la izquierda y una suma. El proceso se describe mejor utilizando un ejemplo. Vamos a multiplicar 2 números binarios: 10111 y 10011, como se muestra en la Figura 8-4. Para realizar la multiplicación, observaremos sucesivamente los bits del multiplicador empezando por el menos significativo. Si el bit del multiplicador es 1, el multiplicando se copia para sumarlo a continuación. Si es 0, se copian tantos ceros como bits tenga el multiplicando. Los números copiados en las líneas siguientes se desplazan una posición a la izquierda, con respecto del número anteriormente copiado, para alinearlos con el bit correspondiente del multiplicador que está siendo procesado. Véase que el producto obtenido de multiplicar dos números binarios de n bit puede llegar a tener un máximo de $2n$ bits, siendo $n \geqslant 2$.

Si se lleva a cabo este procedimiento para multiplicar en hardware digital, es conveniente realizar unos ligeros cambios. Primero, en lugar de utilizar un circuito digital que sume n números binarios simultáneamente, se utilizará un circuito que sume dos números, lo cual es menos costoso. Cada vez que se copia el multiplicando o ceros, estos se sumarán inmediatamente al *producto parcial*. Este producto parcial se almacenará en un registro y quedará preparado para desplazarse a continuación. Segundo, en lugar de desplazar las copias del multiplicando a la izquierda, se desplazará el producto parcial a la derecha. Esto dejará al producto parcial y la copia del multiplicando en la misma posición relativa con el desplazamiento a la izquierda que se hizo del multiplicando. Además, y más importante, en lugar de utilizar un sumador de $2n$ bits, se usará un sumador de n bits. La suma se realizará siempre con las mismas n posiciones, en lugar de moverla un bit a la izquierda cada vez. Tercero, si el bit correspondiente al multiplicador es 0, no se necesitará sumar ceros al producto parcial puesto que esto no altera el valor resultante.

En la Figura 8-5 se repite este ejemplo de multiplicación con estos cambios. Véase que el producto parcial inicial es 0. Cada vez que el bit en proceso del multiplicador es 1 se lleva a cabo una suma con el multiplicando seguida de un desplazamiento a la derecha. Cada vez que el bit del multiplicador es 0 sólo se realiza un desplazamiento a la derecha. Una de estas dos acciones se realiza por cada bit del multiplicador, es este caso, se ejecutan cinco acciones. En azul se indica la aparición de un acarreo durante una de las sumas. Este acarreo no es un problema, sin embargo, el desplazamiento a la derecha que sigue lleva esta información adicional al bit de mayor peso del producto parcial.

23	10111	Multiplicando
19	10011	Multiplicador
	10111	
	10111	
	00000	
	00000	
	10111	
437	110110101	Producto

□ **FIGURA 8-4**
Ejemplo de multiplicación a mano

23	10111	Multiplicando
19	10011	Multiplicador
	00000	Producto parcial inicial
	10111	Suma el multiplicando ya que el bit de multiplicador es 1
	10111	Producto parcial después de sumar y antes de desplazar
	010111	Producto parcial después de desplazar
	10111	Suma el multiplicando ya que el bit es 1
	1000101	Producto parcial después de sumar y antes de desplazar[a]
	1000101	Producto parcial después de desplazar
	01000101	Producto parcial después de desplazar
	001000101	Producto parcial después de desplazar
	10111	Suma multiplicando ya que el bit es 1
	110110101	Producto parcial después de sumar y antes de desplazar
437	0110110101	Producto después del desplazamiento final

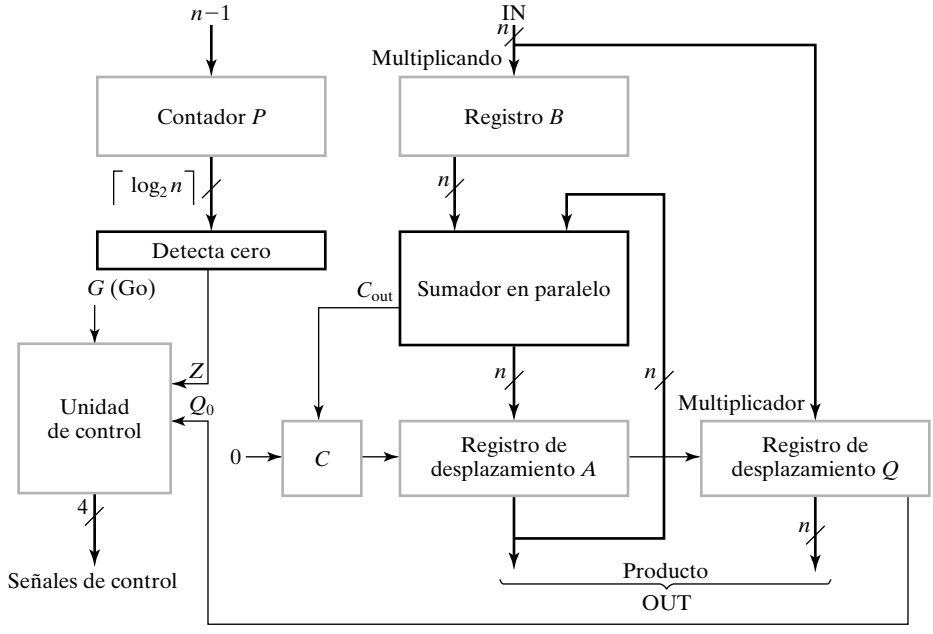
[a] Véase que ocurre un overflow temporalmente

□ **FIGURA 8-5**
Ejemplo de una multiplicación hardware

DIAGRAMA DE BLOQUES DEL MULTIPLICADOR En la Figura 8-6 se muestra el diagrama de bloques del multiplicador binario. La ruta de datos del multiplicador se construye, en un principio, con componentes vistos en capítulos anteriores. Todos, excepto el contador *P*, se han ampliado a *n* bits: el contador necesita $\lceil \log_2 n \rceil$ bits para llevar la cuenta del proceso del multiplicador ($\lceil x \rceil$ es el número entero más pequeño y mayor o igual que x). Usaremos el sumador en paralelo de la Figura 5-5, un registro con carga en paralelo, *B*, similar al de la Figura 7-2, y dos registros de desplazamiento con carga en paralelo, *A* y *Q*, similares a los de la Figura 7-11. El contador *P* es una versión del contador con carga en paralelo de la Figura 7-14, con cuenta descendente en lugar de ascendente, y *C* es un flip-flop con reset y carga síncronos para almacenar C_{out}. Los componentes de esta ruta de datos se muestran en la Figura 8-6.

El multiplicando se carga en el registro *B* procedente de la entrada IN, el multiplicador se carga en el registro *Q*, que también viene por la entrada IN, y el producto parcial se forma en el registro *A* y se almacena en *A* y *Q*. Este doble uso de registro *Q* es posible porque usamos un registro de desplazamiento para el multiplicador *Q* y examinar sucesivamente cada bit del multiplicador que aparezca en Q_0. El desplazamiento a la derecha vacía el bit más significativo del registro *Q*. Este espacio liberado acepta al bit menos significativo del producto parcial de A según se desplaza. El sumador binario de *n* bits se usa para sumar *B* a *A*. El flip-flop *C* almacena el acarreo de la suma, C_{out}, ya sea 0 o 1, y se pone a cero durante el desplazamiento a la derecha. Para contar el número de operaciones suma-desplazamiento que tienen lugar, se utiliza el contador *P*. Inicialmente tiene el valor $n - 1$ y cuenta descendentemente después de obtenerse cada producto parcial. El valor de *P* se comprueba justo antes de su decremento. De esta forma, se realizan *n* operaciones, una para cada valor de *P*, desde $n - 1$ hasta 0. Cada operación es una suma y desplazamiento o sólo un desplazamiento. Cuando *P* alcanza el valor 0, el producto final está ya colocado en el doble registro *A* y *Q* y se termina el proceso.

La unidad de control permanece en un estado inicial hasta que la señal Inicio, *G*, cambia a 1. Entonces el sistema comienza la multiplicación. La suma de *A* y *B* forma los *n* bits más significativos del producto parcial que se transfiere de nuevo a *A*. La salida C_{out} de la suma se transfiere a *C*. Tanto el producto parcial y el multiplicador almacenado en *A* y *Q* se desplazan a



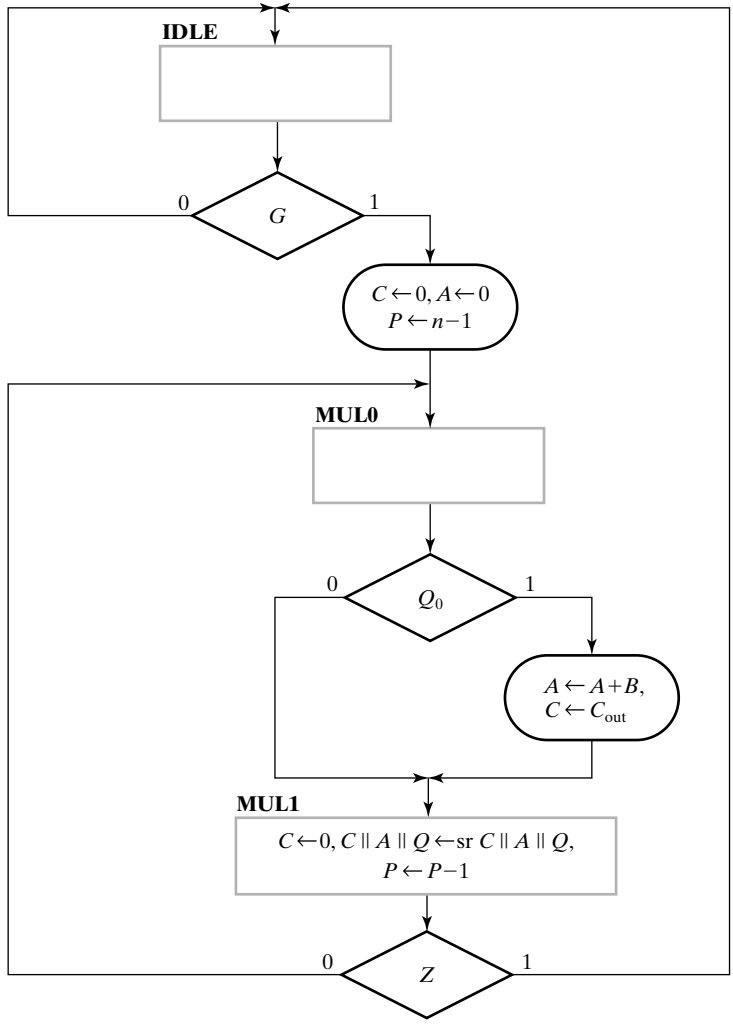
□ **FIGURA 8-6**
Diagrama de bloques para un multiplicador binario

la derecha. El acarreo en C se desplaza al bit más significativo de A, el bit menos significativo de A se desplaza al bit más significativo de Q, y se descarta el bit menos significativo de Q. Después de esta operación de desplazamiento a la derecha, se ha transferido a Q un bit adicional del producto parcial, y los bits del multiplicador se han desplazado una posición a la derecha, de esta forma, el bit menos significativo de Q, Q_0, contiene siempre el bit del multiplicador que la unidad de control examina a continuación. La unidad de control «decide» si sumar, basándose en el valor de este bit. También examina la señal Z, que es 1 si P es igual a cero y 0 si P es distinto de cero, para determinar si se ha terminado la multiplicación. Q_0 y Z son entradas del estado (status) para la unidad de control y la entrada G es la única entrada de control externa. Las señales de control de la unidad de control a la ruta de datos activan las microoperaciones necesarias.

DIAGRAMA ASM DEL MULTIPLICADOR En la Figura 8-7 se muestra la secuencia de operaciones del multiplicador binario mediante un diagrama ASM. Inicialmente, el multiplicando está en el registro B y el multiplicador en Q. La carga de estos registro no se manipula explícitamente por la unidad de control. Mientras el diagrama esté en el estado IDLE y G sea 0, no se producirá ninguna operación y permanecerá en este estado. La multiplicación comienza cuando G cambia a 1. Según el diagrama ASM, para $G = 1$, se mueve del estado IDLE al estado MUL0, los registros C y A se ponen a 0 y el contador se carga con el valor $n - 1$. En el estado MUL0, se decide en base al valor de Q_0, bit menos significativo de Q. Si Q_0 es 1, el contenido de B se suma al de A y el resultado de esta suma se transfiere a A y el acarreo se transfiere a C. Si Q_0 es 0, el registro A y el bit C quedan sin cambiar. En ambos casos el estado futuro es MUL1.

En el estado MUL1 se realiza un desplazamiento a la derecha del contenido de C, A y Q. Este desplazamiento se puede expresar mediante la siguiente lista de cinco transferencias de registros simultáneas:

$$C \leftarrow 0,\ A(n-1) \leftarrow C,\ A \leftarrow \text{sr}A,\ Q(n-1) \leftarrow A(0),\ Q \leftarrow \text{sr}\,Q$$



□ **FIGURA 8-7**
Diagrama ASM para el multiplicador binario

Para simplificar la representación de esta operación, añadimos una notación de bit usando el operador || para definir un registro compuesto construido a partir de otros registros o partes de otros registros. Esta operación, ||, se llama *concatenación*. Por ejemplo,

$$C \,||\, A \,||\, Q$$

representa un registro obtenido a partir de la combinación de registros *C*, *A* y *Q*, desde el más significativo al menos significativo. Podemos usar este registro compuesto para representar el desplazamiento a la derecha

$$C \,||\, A \,||\, Q \leftarrow \text{sr}\, C \,||\, A \,||\, Q$$

como se muestra en la Figura 8-7. Recuerde que suponemos que el bit menos significativo del resultado de un desplazamiento a la derecha toma el valor 0 a no ser que se especifique otra

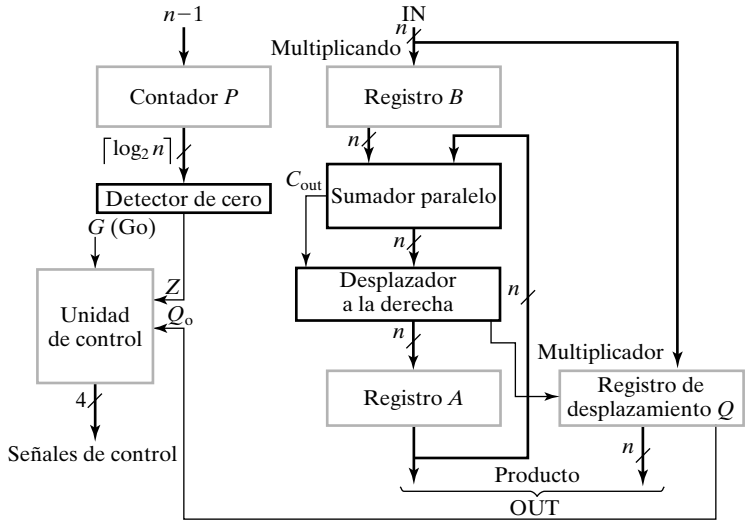
cosa, así que C se pone a 0. Esto se representa explícitamente, no obstante, en el diagrama ASM, ya que C también se pone a 0 en otro estado. La lista de operaciones permite que $C \leftarrow 0$ se realice usando una señal de control en ambos estados.

El contador P se decrementa en el estado MUL1. El valor de P se comprueba en el estado MUL1 antes de decrementar P. Esto ilustra una importante diferencia temporal entre un diagrama de flujo estándar y un diagrama ASM. La decisión en Z, que representa que $P = 0$, sigue a la sentencia de transferencia de registros que actualiza a P en el diagrama ASM. Puesto que la decisión sobre P se lleva a cabo asíncronamente y la sentencia de transferencia de registros es síncrona con el siguiente flanco de subida de reloj, la decisión sobre P precede a la actualización de P. En el siguiente flanco de reloj, cuando P se actualiza, el resultado de esta actualización está disponible para determinar el siguiente estado. En el primer instante $n - 1$ en el que P se comprueba, su contenido no es cero, por lo que el bit de status Z permanece a 0 y el bucle compuesto por los estados MUL0 y MUL1 se ejecuta de nuevo. En el n-ésimo instante de tiempo, el contenido de P es cero y, por tanto, Z es 1. Esto indica que la multiplicación se ha completado haciendo que el ASM vuelva al estado IDLE. El producto final está disponible en $A \,||\, Q$, donde A contiene los n bits más significativos y Q los menos significativos del producto. Merece la pena detenerse un momento para revisar el ejemplo de la multiplicación para $n = 5$ de la Figura 8-5, considerando ahora la relación entre la ruta de datos y el flujo del diagrama ASM.

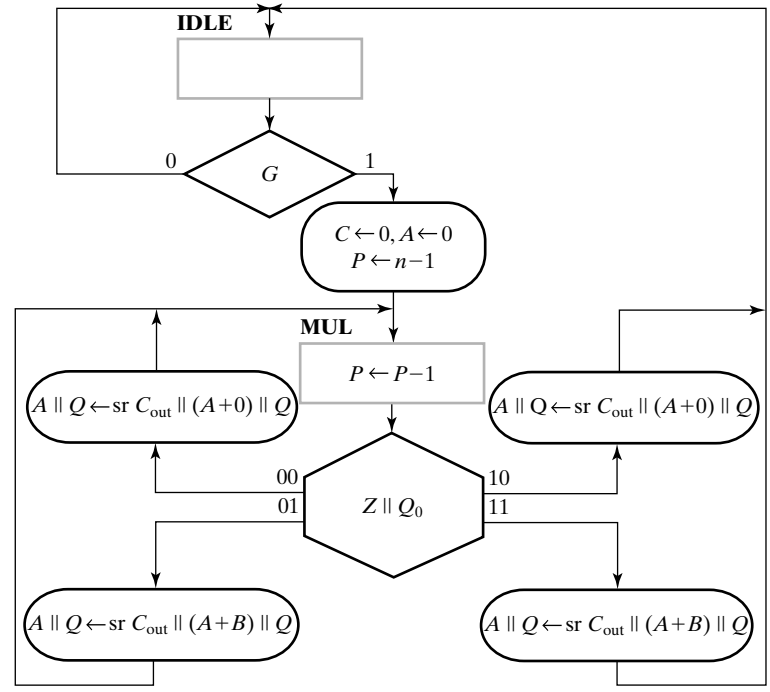
El tipo de registro seleccionado para la ruta de datos puede realizar las microoperaciones enumeradas en el diagrama ASM. El registro A es un registro de desplazamiento con carga en paralelo que guarda el resultado del sumador. También necesita un reset síncrono para poner el registro a 0. El registro Q es un registro de desplazamiento. El flip-flop C almacena el acarreo del sumador y necesita ser puesto a 0 síncronamente. El registro B y Q también deben permitir carga en paralelo para cargar el multiplicando y el multiplicador antes de comenzar el proceso de multiplicación.

La Figura 8-8 muestra otra versión del diseño del multiplicador que usa una caja de decisión vectorial en su diagrama ASM. En la parte (a) de la figura, el registro de desplazamiento se ha sustituido por un desplazador combinacional a la derecha que desplaza un bit en este sentido, similar al de la Figura 5-13(c), y un registro con habilitación de carga como el que se muestra en la Figura 7-2. Esto combina al sumador con un desplazador combinacional a la derecha. Esto nos permite reducir en un estado al diagrama ASM de la Figura 8-8. Para representar el cambio en la ruta de datos del multiplicador es necesario escribir una sentencia de transferencia de registros que combine la suma con el desplazamiento. Además, estos cambios permiten eliminar el flip-flop C. Suponiendo que el retardo del sumador combinado con el desplazador (el cual está hecho sólo con hilos) no es más que el del sumador, la reducción de estados en el bucle de multiplicación incrementa la velocidad de operación de la multiplicación.

Para ilustrar la caja de decisión vectorial, hemos utilizado la concatenación para combinar Z y Q en el vector (Z, Q_0), que se indica como $Z \,||\, Q_0$. La decisión basada en el vector se muestra en el centro del diagrama ASM de la Figura 8-8(b). Hay cuatro combinaciones posibles de salida. Para la combinación en la que Z es igual a 1, el siguiente estado es IDLE. Para la combinación en la que $Z = 0$, es siguiente estado es MUL. Para la combinación en la $Q_0 = 1$, la salida es una suma desplazada a la derecha de los operandos de entrada A y B, y para la combinación en la que $Q_0 = 0$, la salida es una suma desplazada a la derecha de los operandos A y 0. Esto se representa con la transferencia que combina la suma y el desplazamiento en la caja de salida condicional para las cuatro combinaciones de salida de $Z \,||\, Q_0$.



(a)



(b)

□ **FIGURA 8-8**
Multiplicador binario alternativo

8-4 CONTROL CABLEADO

Al diseñar la unidad de control deben considerarse dos aspectos distintos: el control de las microoperaciones y la secuenciación de la unidad de control y las microoperaciones. En pocas palabras, el primero tiene que ver con la parte de control que genera las señales de control, y el

segundo con la parte de control que determina qué sucede después. Aquí separaremos estos dos aspectos dividiendo la especificación original en dos partes: una tabla que define las señales de control en términos de estados y entradas, y un diagrama ASM que solamente representa las transiciones de un estado a otro. Aunque hemos separado estos dos aspectos pensando en el diseño, ambos pueden compartir lógica.

Las señales de control se basan en el diagrama ASM. Las señales de control necesarias para la ruta de datos se enumeran en la Tabla 8-1, donde hemos escogido examinar los registros de la ruta de datos y clasificar las microoperaciones según los registros. Basándonos en dicha clasificación se definen las señales de control. Una de ellas se puede utilizar para activar microoperaciones en más de un registro. Es este caso es razonable puesto que la ruta de datos se emplea para una operación solamente, la multiplicación. De esta forma, las señales de control no necesitar estar separadas para hacer un sistema versátil que permita añadir otras posibles operaciones. Finalmente, la expresión booleana de cada señal de control se extrae de la ubicación de las microoperaciones en el diagrama ASM. Por ejemplo, para el registro *A* hay tres microoperaciones que se indican en la Tabla 8-1: puesta a 0, suma y carga, y desplazamiento a la derecha. Como la operación de puesta a 0 del registro *A* siempre ocurre en el mismo instante de tiempo que la puesta a 0 del flip-flop *C* y la carga del contador *P*, todas ellas pueden activarse con la misma señal de control de inicio, llamada *Initialize*. Aunque *C* también se pone a cero en el estado MUL1, sin embargo, hemos elegido separar sus señales de control. Así, *Initialize* se usa para poner a 0 a *A* y cargar a *P*. En la última columna, para la señal *Initialize*, está la expresión booleana para la que *Initialize* se activa, según se extrae del diagrama ASM, en función del estado IDLE y la entrada *G*. Como *Initialize* se pone a 1 cuando *G* es 1 en el estado IDLE, *Initialize* es el producto lógico de *G* e IDLE. En este punto, el nombre del estado se trata como una variable de estado. Dependiendo de cómo se haga, debe haber una señal que represente al estado o el estado se debe expresar en función de variables de estado. La señal de puesta a cero de *C*, *Clear_C*, se activa en el estado IDLE cuando *G* es igual a 1, de la misma forma que en el

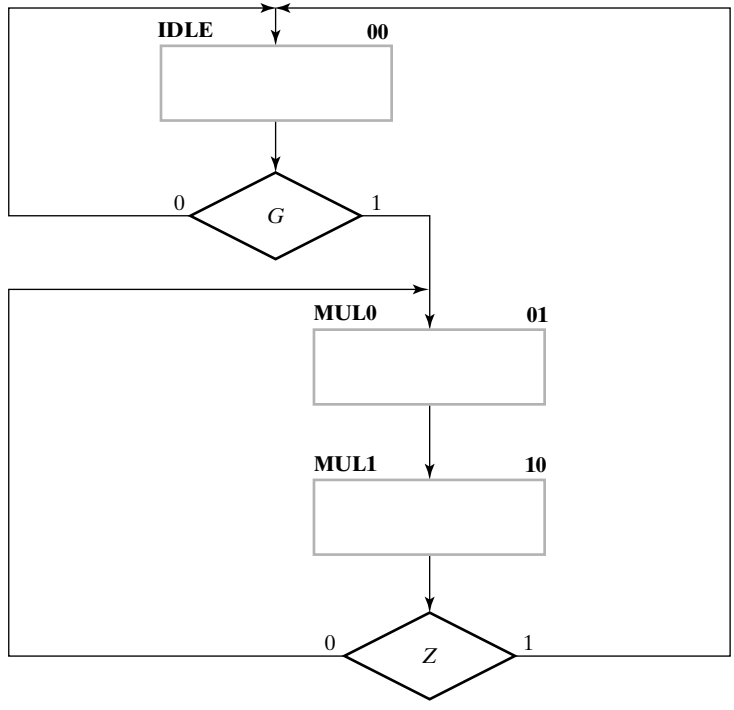
□ **TABLA 8-1**
Señales de control para el multiplicador binario

Módulos del diagrama de bloques	**Microoperaciones**	**Nombre de las señales de control**	**Expresiones de control**
Registro *A*:	$A \leftarrow 0$	Initialize	IDLE · *G*
	$A \leftarrow A + B$	Load	MUL0 · *Q*
	$C \,\|\|\, A \,\|\|\, Q \leftarrow \text{sr}\, C \,\|\|\, A \,\|\|\, Q$	Shift_dec	MUL1
Registro *B*	$B \leftarrow IN$	Load_*B*	*LOADB*
Flip-Flop *C*:	$C \leftarrow 0$	Clear_*C*	IDLE · *G* + MUL1
	$C \leftarrow C_{\text{out}}$	Load	—
Registro *Q*:	$Q \leftarrow IN$	Load_*Q*	*LOADQ*
	$C \,\|\|\, A \,\|\|\, Q \leftarrow \text{sr}\, C \,\|\|\, A \,\|\|\, Q$	Shift_dec	—
Contador *P*:	$P \leftarrow n - 1$	Initialize	—
	$P \leftarrow P - 1$	Shift_dec	—

estado MUL1. Así, se hace el producto lógico de *G* con IDLE y el resultado se suma (operación OR) con MUL1. Las otras dos señales internas de control del multiplicador, *Load* y *Shift_Dec*, se definen de forma similar. Las últimas dos señales, *Load_B* y *Load_Q*, cargan el multiplicando y el multiplicador desde fuera del sistema. Estas señales no se considerarán explícitamente en el resto del diseño.

Eliminando la información relativa a las microoperaciones, podemos volver a dibujar el diagrama ASM representando solamente la información relativa a la secuenciación. El diagrama ASM modificado para el multiplicador binario aparece en la Figura 8-9. Véase que se han eliminado todas las cajas de salidas condicionales. También se han eliminado las cajas de salidas condicionales que no afectan al siguiente estado, en concreto, en la Figura 8-7, la caja de decisión Q_0, que afecta sólo a una caja de salida condicional. Una vez que se ha eliminado la caja de salida condicional, los dos caminos existentes entre la caja de decisión, Q_0, van claramente al mismo estado. Es decir, esta caja de decisión no afecta al siguiente estado y, por tanto, se elimina.

A partir de diagrama ASM modificado, podemos diseñar la parte secuencial de la unidad de control (es decir, la parte que represente el comportamiento del siguiente estado). La división del control en comportamiento del siguiente estado, según el diagrama ASM modificado, y en comportamiento de salida, según la tabla de señales de control, indica como el ASM relaciona el siguiente estado y la parte de salida del circuito secuencial. La Figura 8-9 se corresponde con el diagrama de estados de un circuito secuencial sin las salidas especificadas, exceptuando que la representación de los estados y las transiciones son distintas. Debido a esta relación, podemos tratar el diagrama ASM como un diagrama de estados y crear una tabla para la parte de secuenciación de la unidad de control. Posteriormente, la unidad de control se puede diseñar mediante



□ **FIGURA 8-9**
Diagrama ASM para la parte de secuenciación del multiplicador binario

métodos de diseño lógico secuencial, como se planteó en el Capítulo 4. Sin embargo, en muchos casos, es difícil llevar a cabo este método debido al gran número de estados que típicamente tiene una unidad de control. Como consecuencia, usaremos métodos específicos para el diseño de unidades de control, que son variaciones de los métodos clásicos de diseño secuencial. Presentamos e ilustramos a continuación dos de estos métodos de diseño.

Registro de secuencia y decodificador

El método de registro de secuencia y decodificador, como su nombre indica, usa un registro de secuencia para el control de estados y un decodificador que proporciona una señal de salida que se corresponde con cada estado. Un registro con n flip-flops puede tener hasta 2^n estados y un decodificador n a 2^n tiene un máximo de 2^n salidas, una para cada uno de los estados. Un registro de secuencia es, esencialmente, n flip-flops juntos con puertas asociadas que afectan a las transiciones entre sus estados. Puede ser necesario añadir lógica para el control de las señales de salida.

La parte secuencial de un diagrama ASM para el multiplicador binario tiene tres estados y dos entradas. Para realizar el diagrama ASM con un registro de secuencia y decodificador hacen falta dos flip-flops para el registro y un decodificador de 2 a 4 líneas. Al tener tres estados, solamente se usan tres de la cuatro salidas del decodificador. Aunque este es un ejemplo sencillo, el procedimiento planteado también se aplica en situaciones más complejas.

La tabla de estados para la parte de secuenciación de la unidad de control se muestra en la Tabla 8-2. Llamaremos a los dos flip-flops como M_1 y M_0 y asignaremos los estados binarios 00, 01 y 10 a IDLE, MUL0 y MUL1, respectivamente. Véase que las columnas de las entradas tiene entradas sin especificar ($\times$) siempre que una variable de entrada no se use para determinar el siguiente estado. Las salidas de la parte de secuenciamiento del control se denominan mediante el nombre de los estados. El código binario para el estado actual determina la variable de salida que es igual a 1 en un instante dado. De esta forma, cuando el estado actual es $M_1M_0 = 00$, la salida IDLE es igual a 1 mientras que el resto de las salidas son iguales a 0. Como las salidas dependen sólo del estado actual, se pueden generar con un decodificador de 2 a 4 líneas, que tiene como entradas M_1 y M_0 y como salidas IDLE, MUL0 y MUL1.

□ **TABLA 8-2**
Tabla de estados para el registro de secuencia y el decodificador de la unidad de control del multiplicador

Estado actual			Entradas		Estado futuro		Salidas del decodificador		
Nombre	M_1	M_0	G	Z	M_1	M_0	**IDLE**	**MUL0**	**MUL1**
IDLE	0	0	0	×	0	0	1	0	0
	0	0	1	×	0	1	1	0	0
MUL0	0	1	×	×	1	0	0	1	0
MUL1	1	0	×	0	0	1	0	0	1
	1	0	×	1	0	0	0	0	1
—	1	1	×	×	×	×	×	×	×

Como se mencionó anteriormente, el circuito secuencial se puede diseñar a partir de la tabla de estados usando el método de diseño de circuito secuenciales presentado en el Capítulo 4. Este ejemplo tiene un número pequeño de estados y de entradas, y podríamos, por tanto, usar Mapas de Karnaugh para simplificar las funciones booleanas. Sin embargo, en la mayoría de aplicaciones el número de estados es mucho mayor. La aplicación del método convencional requiere un trabajo excesivo para obtener las ecuaciones simplificadas para los flip-flops. Aquí, el diseño se puede simplificar si tenemos en consideración el hecho de que las salidas del decodificador están disponibles para su uso en el diseño. En lugar de usar las salidas de los flip-flops como las condiciones del estado actual, podríamos también usar las salidas del decodificador para obtener esta información. Estas salidas proporcionan una sola señal que representa cada uno de los posibles estados del circuito. Es más, en lugar de usar los Mapas de Karnaugh para simplificar las ecuaciones de los flip-flops, las podemos obtener por simple inspección de la tabla de estados. Por ejemplo, a partir de las condiciones del estado futuro de la tabla, encontramos que el estado futuro de M_0 es igual a 1 si el estado actual es IDLE y la entrada G es igual a 1 o cuando el estado presente es MUL1 y su entrada Z es igual a 0. De estas condiciones se deduce que:

$$D_{M_0} = \text{IDLE} \cdot G + \text{MUL1} \cdot \overline{Z}$$

para la entrada D del flip-flop M_0. De la misma forma, la entrada D del flip-flop M_1 es:

$$D_{M_1} = \text{MUL0}$$

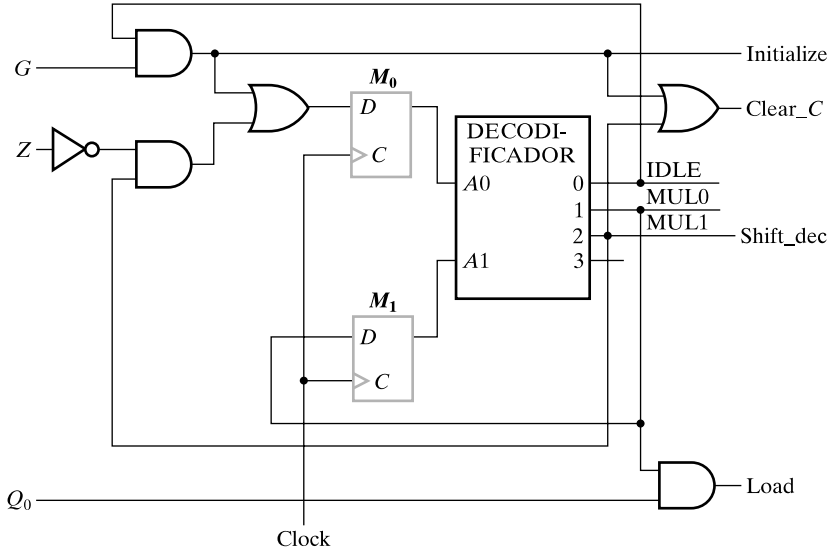
Véase que estas ecuaciones se han extraído directamente de inspeccionar la tabla que utiliza los nombres de los estados en lugar del nombre de las variables de estado puesto que está el decodificador que proporciona los símbolos de los estados. En algunos casos es posible encontrar ecuaciones más simples para las entradas D de los flip-flops usando variables de estado directamente en lugar de los estados. Podemos eliminar redundancia y reducir costes escribiendo las ecuaciones del decodificador y aplicar una herramienta de simplificación al conjunto de las ecuaciones de control.

El diagrama para el control aparece en la Figura 8-10. Está formado por un registro de 2 bits con los flip-flops M_1 y M_0 y un decodificador de 2 a 4 líneas. Tres salidas del decodificador se usan para generar las salidas de control, así como las entradas para la lógica que calcula el estado futuro. Las salidas *Initialize*, *Clear_C*, *Shift_dec* y *Load* se determinan a partir de la Tabla 8-1. *Initialize* y *Shift_dec* ya están disponibles como señales y se han añadido como líneas de salidas etiquetadas. Sin embargo, como se muestra en la figura, tenemos que añadir puertas lógicas para *Clear_C* y *Load*. Completamos el diseño del multiplicador binario conectando las salidas de la unidad de control a las entradas de control de la ruta de datos.

Un flip-flop por estado

Otro posible método de diseño de la lógica de control es usar un flip-flop por estado. Cada flip-flop se asigna a un estado y, para cada instante de tiempo, sólo uno de los flip-flops estará a uno y el resto a 0. Cuando un flip-flop asignado a un estado concreto está a 1 el circuito secuencial está en ese mismo estado. Este único 1 se propaga de un flip-flop a otro según la lógica de decisión del control. Para dicha configuración, cada flip-flop representa al estado actual sólo cuando ese único 1 se almacena en dicho flip-flop.

Es evidente que, excepto en algunas técnicas de detección y corrección de errores, este método usa el máximo número de flip-flops posibles en un circuito secuencial. Por ejemplo, un



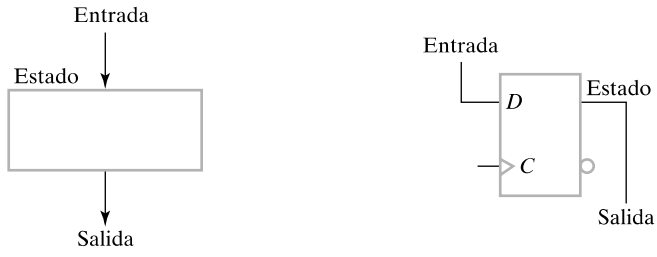
□ **FIGURA 8-10**
Unidad de control para un multiplicador con un registro de secuencia y un decodificador

circuito secuencial con 12 estados, que use un número mínimo de variables de estado codificadas, necesita sólo cuatro flip-flops. Con un flip-flop por estado, el circuito necesita 12 flip-flops, uno por estado. A primera vista puede parecer que este método incrementaría el coste del sistema puesto que utiliza más flip-flops. Pero el método ofrece algunas ventajas en este aspecto que pueden no ser aparentes. Una ventaja es la simplicidad con la que su lógica se puede diseñar, simplemente por inspección del diagrama ASM o el diagrama de estados. Si se emplean flip-flops tipo *D*, no se necesitan tablas de estados ni de excitación. Esto da lugar a un ahorro en el esfuerzo de diseño.

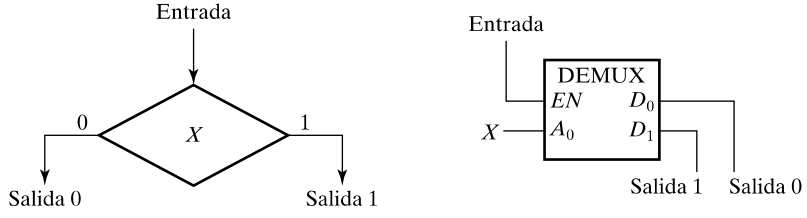
En la Figura 8-11 se muestra las reglas de sustitución para transformar un diagrama ASM a un circuito secuencial con un flip-flop por estado. Estas reglas se utilizan con más facilidad en un diagrama ASM que sólo represente la secuenciación de información, tal y como se muestra en la Figura 8-9. Cada regla especifica la sustitución de un componente de un diagrama ASM con un circuito lógico. Como se muestra en la Figura 8-11(a), la caja de estado se sustituye por un flip-flop de tipo *D* etiquetado con el mismo nombre que el estado. La entrada de la caja de estado se corresponde con la entrada *D* del flip-flop. La salida de la caja de estado se corresponde con la salida del flip-flop.

En la Figura 8-11(b), la caja de decisión escalar se reemplaza por un demultiplexor de 2 salidas. La señal correspondiente a la entrada de la caja de decisión se manda a una de las dos líneas existentes, dependiendo del valor de la señal *X*. Si *X* es 0, la señal se manda a la línea de salida 0; si *X* es 1, la señal se manda a la línea de salida 1. De esta forma, en el ejemplo, si hay un 1 en el circuito que está en la entrada de la caja de decisión, y *X* es 0, el 1 pasa a la línea de salida 0. El demultiplexor actúa como un conmutador que lleva el 1 a través de los caminos del circuito que se corresponden a los caminos del diagrama ASM.

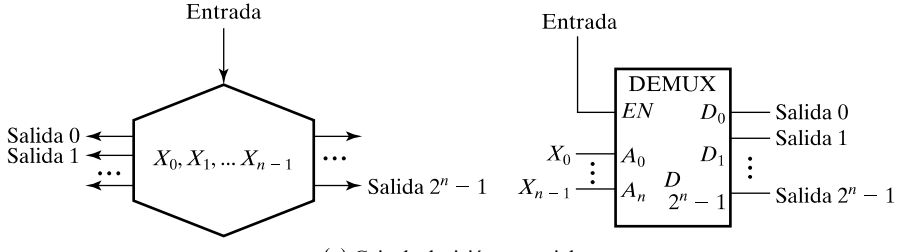
En la Figura 8-11(c), la caja de decisión vectorial se sustituye por un demultiplexor de *n* entradas. La señal correspondiente a la entrada de la caja de decisión se manda a una de las $2^n - 1$ líneas, dependiendo del valor del vector $X = X_0, ..., X_{n-1}$. Si $X = 0$, la señal se envía a la línea de salida 0; si *X* es 9, la señal se envía a la línea de salida 9. En el ejemplo, si el único 1



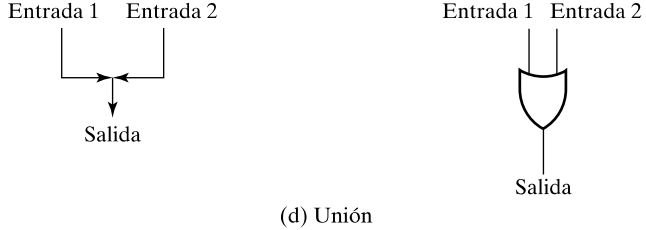
(a) Caja de estado



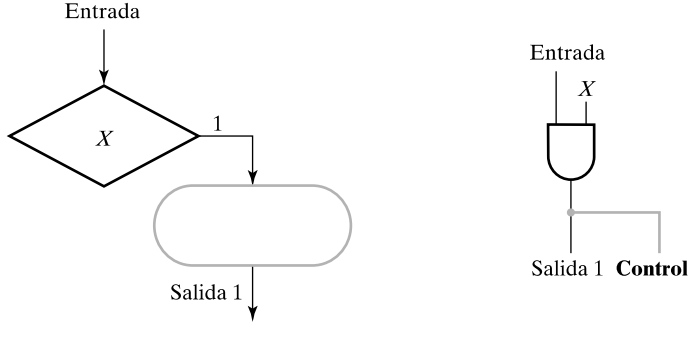
(b) Caja de decisión escalar



(c) Caja de decisión vectorial



(d) Unión



(e) Caja de salida condicional

□ **FIGURA 8-11**
Reglas de transformación para la unidad de control con un flip-flop por estado

del circuito está a la entrada de la caja de decisión y *X* es 9, el 1 se pasa a la línea de salida 9. El demultiplexor actúa como un conmutador que direcciona el 1 a través de los caminos del circuito correspondientes a los caminos del diagrama ASM.

La unión de la Figura 8-11(d) se hace en cualquier punto en el cual dos o más líneas direccionadas en el diagrama ASM se unen. Si hay en el circuito un 1 en cualquier línea correspondiente a uno de los caminos de entrada, entonces debe aparecen en la línea correspondiente al camino de salida, dando a la línea el valor 1. Si ninguna de las líneas correspondientes a los caminos de entrada de la unión tiene un 1, la línea de salida debe tener el valor 0. Por esto, la unión se sustituye por una puerta OR.

Con estas cuatro transformaciones, la parte de secuenciamiento del diagrama ASM puede reemplazarse por un circuito con un flip-flop por estado por simple inspección. A la hora de manejar las salidas, sólo es cuestión de conectar las líneas de control al sitio adecuado del circuito o añadiendo lógica a la salida. Las salidas se basan en el diagrama original ASM o en la tabla de señales de control derivadada del diagrama. La conexión de una línea de control según un diagrama ASM se ilustra mediante la caja de salida condicional de la Figura 8-11(e). La caja de salida condicional en el diagrama ASM se sustituye simplemente por una conexión al circuito. Pero para hacer que sucedan cambios a la salida, se saca una línea de control desde la conexión y se etiqueta con la variable de salida. Por claridad, dicha transformación se muestra en azul.

Ahora usaremos estas transformaciones para diseñar la unidad de control con un bit por estado para el multiplicador binario.

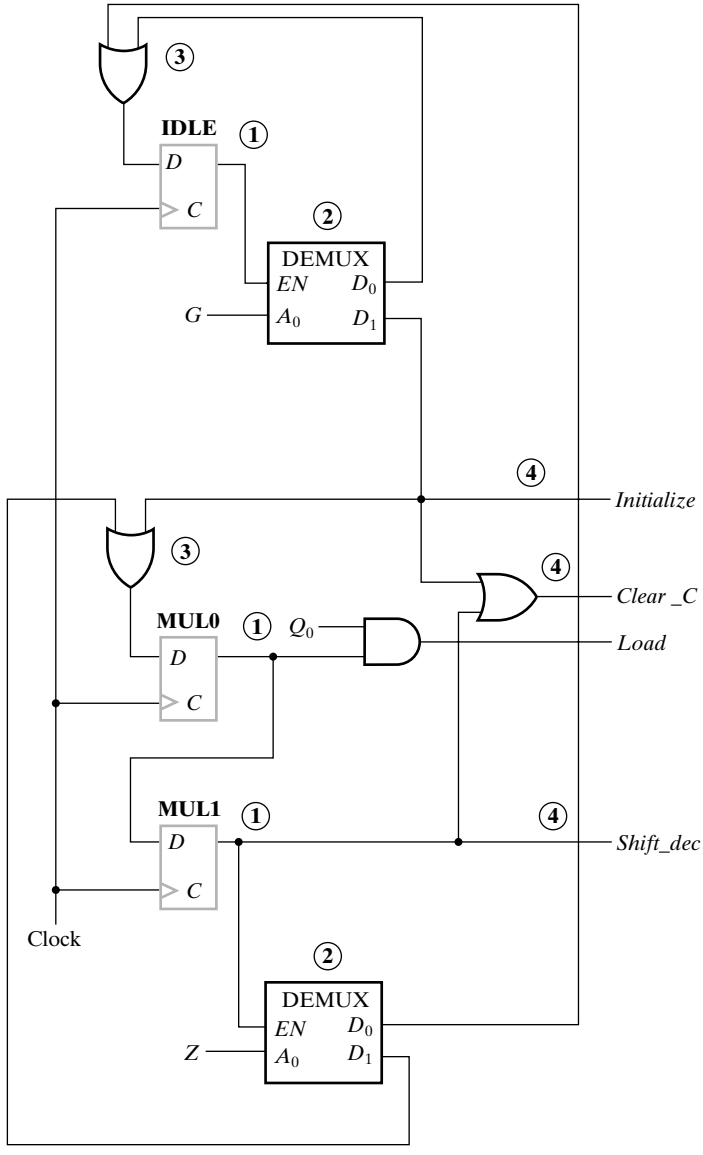
EJEMPLO 8-1 Multiplicador binario

El diagrama ASM de la Figura 8-9 se usará para el diseño de la parte de secuenciamiento. Véase que se ignoran los códigos binaros dados puesto que son del método de diseño anterior. El diagrama lógico resultante se muestra en la Figura 8-12.

Primero reemplazamos cada una de las tres cajas de estados por flip-flops tipo *D* etiquetados con el nombre del estado, como se indica en los círculos con el número 1 de la figura. Segundo, cada una de las cajas de decisión se sustituye por un demultiplexor con la variable de decisión como entrada de selección, según se indica con los círculos con el número 2 de la figura. Tercero, cada unión se reemplaza por una puerta OR, como se indica en los círculos con el número 3 de la figura. Finalmente, las conexiones representadas por flechas en el diagrama ASM se añaden desde las salidas a las entradas de los componentes correspondientes.

Para manejar las salidas de control, podemos usar la Tabla 8-1 o el diagrama ASM original de la Figura 8-7. Partiendo de la tabla, vemos que la función booleana para Initialize ya está en el diagrama lógico, así que simplemente añadimos la salida etiquetada como *Initialize*. De la misma forma, se puede añadir la salida *Shift_dec*. Para las salidas *Clear_C* y *Load* es necesario añadir puertas lógicas. Todas las conexiones de salida y la lógica adicional se han indicado con los círculos con el número 4 en la Figura 8-12.

La última tarea a realizar en el diseño de la lógica de control con un flip-flop por estado es la de inicialización de estados, poniendo un 1 al flip-flop IDLE y 0 a los demás. Esto se puede realizar mediante una entrada de PRESET asíncrono en el flip-flop IDLE y entradas CLEAR asíncronas en los otros flip-flops. Si sólo disponemos de flip-flops con entrada de CLEAR asíncrono en lugar de tener las dos entradas de CLEAR y PRESET, se puede colocar un inversor a la entrada *D* y otro a la salida del flip-flop IDLE. Así el flip-flop IDLE contendrá un 0 en el estado IDLE y un 1 es el resto de estados. Esto permite utilizar un CLEAR asíncrono para



□ **FIGURA 8-12**
Unidad de control con un flip-flop por estado para el multiplicador binario

inicializar los tres flip-flops del circuito. Se debe tener en cuenta que, aparte de para inicializar el circuito, el uso de las entradas asíncronas de los flip-flops para realizar diagramas ASM u otros circuitos secuenciales es, generalmente, desaconsejable. ■

Una vez que se ha completado el diseño básico de la lógica de control, sería deseable refinar el diseño. Por ejemplo. si hay un número de uniones conectadas mediante líneas, las puertas OR resultantes de la transformación se pueden combinar. Los demultiplexores puestos es cascada unos con otros también se pueden combinar. Otro tipo de simplificación o mapeo tecnológico se puede aplicar al diseño.

8–5 REPRESENTACIÓN HDL DEL MULTIPLICADOR BINARIO-VHDL

El multiplicador binario estudiado se puede modelar en la etapa de diseño mediante una descripción de comportamiento VHDL. En las Figuras 8-13 y 8-14 aparece la descripción de una versión de 4 bits del multiplicador. Este código VHDL representa al diagrama de bloques de la Figura 8-6 y al diagrama ASM de la Figura 8-7. El código VHDL está formado por la entidad `binary_multiplier` y la arquitectura `behavior_4`. La arquitectura contiene dos sentencias de asignación y tres procesos. Los procesos son similares a los usados en detector de secuencia del Capítulo 6. La principal diferencia es que el proceso de la función de salida se ha sustituido por un proceso que describe la transferencia de registros de la ruta de datos. Debido a este cambio, la representación VHDL se corresponde más con la descripción de la Tabla 8-1 y del diagrama ASM de la Figura 8-9 que con la del diagrama de la Figura 8-7.

En la entidad se definen las entradas y las salidas del multiplicador. Al comienzo de la arquitectura, una declaración de tipos define los tres estados. Las señales internas, algunas de las cuales se generara en los registros, se declaran a continuación. Entre estas están `state` y `next_state` para el control, los registros `A`, `B`, `P` y `Q` y el flip-flop `C`. Por conveniencia también se declara la señal `Z`. A continuación se hace una asignación que fuerza `Z` a 1 siempre que `P` tenga el valor 0. Posterior a esto, las salidas concatenadas de los registros `A` y `Q` se asignan a la salida del multiplicador `MULT_OUT`. Esto es necesario, en lugar de hacer a `A` y a `Q` salidas del circuito, para permitir usar `A` y `Q` dentro del circuito.

El resto de la descripción está compuesto por tres procesos. El primer proceso describe el registro de estados e incluye un `RESET` y el reloj. El segundo describe la función que calcula el estado futuro a partir de la Figura 8-8. Dese cuenta que, aunque el reloj y el `RESET` se incluyen en el registro de estados, no aparecen en la figura. En la lista de sensibilidad se incluyen todas las señales que pueden afectar al siguiente estado. Por otra parte, este proceso se parece al proceso `next_state` del detector de secuencia.

El proceso final de la Figura 8-14 describe la función de la ruta de datos. Como las condiciones para llevar a cabo una operación se definen en función de los estados y de las entradas. Este proceso también define implícitamente las señales de control dadas en la Tabla 8-1. Sin embargo, estas señales de control no aparecen explícitamente. Puesto que la función de la ruta de datos tiene registros como destino de todas las asignaciones, todas las transferencias se controlan con `CLK`. Como los datos se cargarán siempre en estos registros antes de la multiplicación, no es necesario que éstos tengan una señal de reset. La primera sentencia **`if`** controla la carga del multiplicando en el registro `B` y la segunda sentencia **`if`** controla la carga del multiplicador en el registro `Q`.

Las transferencias de registros directamente involucrados en la multiplicación se controlan mediante una sentencia **`case`** que depende de estado del control, la entrada `G`, y las señales internas `Q(0)` y `Z`. Estas transferencias se plantean en la Figura 8-7 y en la Tabla 8-1. El modelado de la suma en el estado `MUL0` requiere algo de esfuerzo. Antes de nada, para llevar a cabo la suma con vectores `std_logic`, aparece una sentencia **`use`** justo antes de la declaración de entidad para el paquete `ieee.std_logic_unsigned.`**`all`**. Además, para la suma necesitamos transferir el acarreo de salida, C_{out}, de la suma a *C*. Para conseguirlo, realizamos una suma de 5 bits añadiendo ceros a la izquierda de `A` y `B`, y asignando el resultado a una variable de 5 bits, `CA`. Una alternativa podría ser escribir `C & A` como transferencia de destino después de la sentencia if, pero el uso de la concatenación, `&`, en los registros de destino no se permite en VHDL. Al ser `CA` una variable, su valor se asigna inmediatamente y está disponible para la asignación

```
-- Multiplicador binario con n = 4: Descripción VHDL
-- véase Figuras 8-6 y 8-7 del diagrama de bloques y ASM
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity binary_multiplier is
  port(CLK, RESET, G, LOADB, LOADQ: in std_logic;
    MULT_IN: in std_logic_vector(3 downto 0);
    MULT_OUT: out std_logic_vector(7 downto 0));
end binary_multiplier;

architecture behavior_4 of binary_multiplier is
  type state_type is (IDLE, MUL0, MUL1);
  signal state, next_state : state_type;
  signal A, B, Q: std_logic_vector(3 downto 0);
  signal P: std_logic_vector(1 downto 0);
  signal C, Z: std_logic;
begin
  Z <= P(1) NOR P(0);
  MULT_OUT <= A & Q;

  state_register: process (CLK, RESET)
  begin
    if (RESET = '1') then
      state <= IDLE;
    elsif (CLK'event and CLK = '1') then
      state <= next_state;
    end if;
  end process;

  next_state_func: process (G, Z, state)
  begin
    case state is
      when IDLE =>
        if G = '1' then
          next_state <= MUL0;
        else
          next_state <= IDLE;
        end if;
      when MUL0 =>
        next_state <= MUL1;
      when MUL1 =>
        if Z = '1' then
          next_state <= IDLE;
        else
          next_state <= MUL0;
        end if;
```

□ **FIGURA 8-13**
Descripción VHDL de un multiplicador binario

de `C` y `A` después de la sentencia **`if`**. En el estado `MUL1`, el desplazamiento se realiza usando la concatenación, como se hizo en el ejemplo del Capítulo 5. *P* se decrementa restando una constante de 2 bit de valor 1.

```
    end case;
  end process;

  datapath_func: process (CLK)
  variable CA: std_logic_vector(4 downto 0);
  begin
    if (CLK'event and CLK = '1') then
      if LOADB = '1' then
        B <= MULT_IN;
      end if;
      if LOADQ = '1' then
        Q <= MULT_IN;
      end if;
      case state is
        when IDLE =>
          if G = '1' then
            C <= '0';
            A <= "0000";
            P <= "11";
          end if;
        when MUL0 =>
          if Q(0) = '1' then
            CA := ('0' & A) + ('0' & B);
          else
            CA := C & A;
          end if;
            C <= CA(4);
            A <= CA(3 downto 0);
        when MUL1 =>
          C <= '0';
          A <= C & A(3 downto 1);
          Q <= A(0) & Q(3 downto 1);
          P <= P - "01";
      end case;
    end if;
  end process;
end behavior_4;
```

□ **FIGURA 8-14**
Descripción VHDL de un multiplicador binario (*continuación*)

Esta descripción se puede simular para validarla y sintetizar automáticamente para generar la lógica, si se desea.

8-6 REPRESENTACIÓN HDL DEL MULTIPLICADOR BINARIO-VERILOG

El multiplicador estudiado se puede modelar durante la etapa de diseño como una descripción de comportamiento en Verilog. En las Figuras 8-15 y 8-16 se describe una versión de 4 bits del multiplicador. Este código en Verilog representa al diagrama de bloques de la Figura 8-6 y al diagrama ASM de la Figura 8-7. Este código está compuesto de un módulo llamado

```
// Multiplicador binario con n = 4: Descripción Verilog
// véase Figuras 8-6 y 8-7 del diagrama de bloques y ASM

module binary_multiplier_v (CLK, RESET, G, LOADB, LOADQ,
       MULT_IN, MULT_OUT);
input CLK, RESET, G, LOADB, LOADQ;
input [3:0] MULT_IN;
output [7:0] MULT_OUT;
reg [1:0] state, next_state, P;
parameter IDLE = 2'b00, MUL0 = 2'b01, MUL1 = 2'b10;
reg [3:0] A, B, Q;
reg C;
wire Z;

assign Z = ~| P;
assign MULT_OUT = {A,Q};

//state register
always@(posedge CLK or posedge RESET)
begin
  if (RESET == 1)
    state <= IDLE;
  else
  state <= next_state;
end

//next state function
always@(G or Z or state)
begin
  case (state)
    IDLE:
      if (G == 1)
        next_state <= MUL0;
      else
        next_state <= IDLE;
    MUL0:
      next_state <= MUL1;
    MUL1:
      if (Z == 1)
        next_state <= IDLE;
      else
        next_state <= MUL0;
  endcase
end

//datapath function
always@(posedge CLK)
```

□ **FIGURA 8-15**
Descripción Verilog de un multiplicador binario

`binary_multiplier_v`. La descripción contiene dos sentencias de asignación y tres procesos. Los procesos son similares a los que se usaron para el detector de secuencia del Capítulo 6. La diferencia principal es que el proceso de la función de salida se ha sustituido por un proceso

```
begin
  if (LOADB == 1)
    B <= MULT_IN;
  if (LOADQ == 1)
    Q <= MULT_IN;
  case (state)
    IDLE:
      if (G == 1)
        begin
          C <= 0;
          A <= 4'b0000;
          P <= 2'b11;
        end
    MUL0:
      if (Q[0] == 1)
        {C, A} = A + B;
    MUL1:
      begin
        C <= 1'b0;
        A <= {C, A[3:1]};
        Q <= {A[0], Q[3:1]};
        P <= P - 2'b01;
      end
  endcase
end
endmodule
```

□ **FIGURA 8-16**
Descripción Verilog de un multiplicador binario (*continuación*)

que describe la transferencia de registros de la ruta de datos. Debido a este cambio, la representación en Verilog se corresponde más a la descripción de la Tabla 8-1 y al diagrama de la Figura 8-9 que al diagrama ASM de la Figura 8-7.

Al comienzo de la descripción se definen las salidas y las entradas del multiplicador. Una declaración de parámetro define los tres estados y sus códigos binarios. Se definen las señales de tipo registro. Entre estas están `state` y `next_state` para el control, los registros `A`, `B`, `P` y `Q`, y el flip-flop `C`. Según las especificaciones de reloj, casi todos los registros se actualizarán con el flanco de subida. La excepción a destacar es `next_state`. También se ha declarado por conveniencia la señal intermedia `Z` de tipo *wire*. A continuación se hace una asignación que fuerza a `Z` a ser 1 siempre que `P` contenga el valor 0. Esta sentencia usa la operación OR (|) como *operador de reducción*. La reducción es la aplicación de un operador a un hilo (*wire*) o a un registro que combina bits individualmente. En este caso, la aplicación de la operación OR a *P* produce la suma lógica de todos los bits de *P* juntos. Al estar la operación precedida por ∼, la operación lógica resultante es una NOR. Se pueden aplicar otros operadores como operadores de reducción. La segunda sentencia de asignación asigna las salidas de los registros concatenados `A` y `Q` a la salida del multiplicador `MULT_OUT`. Esto se hace por conveniencia para hacer de la salida una estructura única.

El resto de la descripción está compuesta de tres procesos. El primer proceso describe al registro de estado e incluye un `RESET` y un reloj. El segundo proceso describe la función que calcula el estado futuro de la Figura 8-9. Tenga en cuenta que, aunque el reloj y el `RESET` se incluyen en el registro de estados, éstos no aparecen en la figura. En la sentencia de control de

eventos se incluyen todas las señales que pueden afectar al siguiente estado. Estas son `G`, `Z` y `state`. Por otra parte, este proceso se parece al proceso que calcula el estado futuro del detector de secuencia.

El último proceso describe la función de la ruta de datos. Como las condiciones para realizar una operación se define en función de los estados y las entradas, este proceso también define implícitamente las señales de control dadas en la Tabla 8-1. Sin embargo, estas señales de control no aparecen explícitamente. Puesto que la función de la ruta de datos tiene registros como destino de todas las asignaciones, todas las transferencias se controlan con `CLK`. Como los datos se cargarán siempre en estos registros antes de la multiplicación, no es necesario que estos tengan una señal de un reset. La primera sentencia **`if`** controla la carga del multiplicando en el registro `B` y la segunda sentencia **`if`** controla la carga del multiplicador en el registro `Q`.

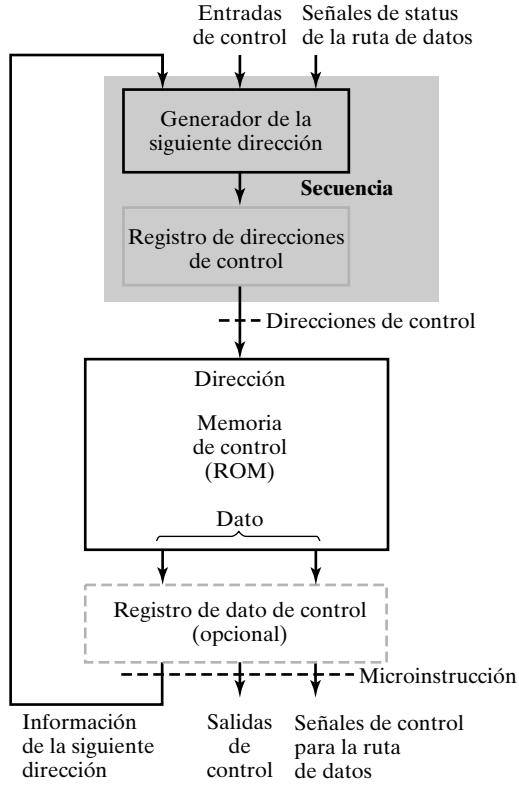
Las transferencia de registros directamente involucradas en la multiplicación se controlan mediante una sentencia **`case`** que depende del estado de control, la entrada `G`, y las señales internas `Q(0)` y `Z`. Estas transferencias se plantean en la Figura 8-7 y en la Tabla 8-1. La representación de la suma en el estado `MUL0` usa la concatenación de `C` y `A` para obtener el acarreo de salida, C_{out}, y cargarlo en *C*. Verilog permite la suma de dos operando de 4 bits y dar el resultado con 5 bits. En el estado `MUL1`, el desplazamiento se lleva a cabo usando la concatenación como se hizo en el ejemplo del Capítulo 5. `P` se decrementa restando una constante de 2 bits de valor 1.

Esta descripción se puede simular para validarla y sintetizar automáticamente para generar la lógica, si se desea.

8–7 CONTROL MICROPROGRAMADO

Un *control microprogramado* es una unidad de control que tiene almacenados sus valores binarios de control como palabras en una memoria. Cada palabra de la memoria de control contiene una *microinstrucción* que especifica una o más microoperaciones de un sistema. Una secuencia de microinstrucciones constituyen un *microprograma*. El microprograma se ajusta durante el diseño del sistema y se almacena en una ROM. La microprogramación involucra la colocación en la ROM de las combinaciones que representan a los valores de las variables de control en palabras. A estas representaciones se accede mediante operaciones de lecturas sucesivas y se usarán por el resto de la lógica de control. Los valores de una palabra de ROM de una determinada dirección especifican las operaciones que se llevarán a cabo tanto en la ruta de datos como en la unidad de control. Un microprograma también se puede almacenar en una RAM. Es este caso, se carga al iniciarse el sistema desde alguna forma de almacenamiento no volátil, por ejemplo, un disco magnético. La memoria de la unidad de control, ya sea RAM o ROM se llama *memoria de control*. Si se usa una RAM, la llamaremos *memoria de control de escritura*.

En la Figura 8-17 se muestra la configuración general de un control microprogramado. Suponiendo que la memoria de control es una ROM, todos los microprogramas están almacenados permanentemente. El *registro de direcciones de control* (*CAR*, del inglés *control address register*) especifica la dirección de la microinstrucción. El *registro de datos de control* (*CDR*, del inglés *control data register*), que es opcional, puede contener la microinstrucción en curso que se ejecuta en la ruta de datos y en la unidad de control. Una de las funciones de la palabra de control es determinar la dirección de la siguiente microinstrucción a ejecutar. Esta microinstrucción puede ser la siguiente de la secuencia o puede estar localizada en algún otro sitio de la memoria. Por tanto, uno o más bits de la microinstrucción en curso pueden especificar la forma de determinar la dirección de la siguiente microinstrucción. La siguiente dirección también puede ser función del status y de las entradas externas de control. Cuando se ejecuta una mi-



□ **FIGURA 8-17**
Organización de la unidad de control microprogramado

croinstrucción, el *generador de la siguiente dirección* produce la siguiente instrucción. Esta instrucción se transfiere al CAR en el siguiente pulso de reloj y se utiliza para leer la siguiente microinstrucción a ejecutar de la ROM. De este modo, las microinstrucciones contienen bits que activan las microoperaciones de la ruta de datos y los bits que especifican la secuencia de microinstrucciones a ejecutar.

Al generador de la siguiente dirección, en combinación con el CAR, se le suele denominar *secuenciador* de programa ya que determina la secuencia de instrucciones de lectura de la memoria de control. La dirección de la siguiente microinstrucción se puede especificar de varias formas, dependiendo de las entradas del secuenciador. Las funciones típicas del secuenciador del microprograma son: incremento en uno del CAR y carga del CAR. Las posibles fuentes para la operación de carga son: una dirección de la memoria de control, una dirección externa y una dirección inicial para empezar la operación de la unidad de control.

El CDR contiene la microinstrucción actual mientras que se calcula la siguiente dirección y se lee la siguiente microinstrucción de la memoria. El CDR reduce los retardos de los caminos combinacionales que van a través de la memoria de control y siguen hacia la ruta de datos. Su presencia permite al sistema operar a mayor frecuencia de reloj y, por tanto, procesar la información más rápidamente. Sin embargo, la inclusión del CDR en un sistema complica la secuenciación de las microinstrucciones, en especial cuando las decisiones a tomar dependen de los bits de status. Para simplificar esta breve introducción, omitiremos el CDR y tomaremos las microinstrucciones directamente de las salidas de la ROM. La ROM funciona como un circuito

combinacional, con las direcciones como entradas y la correspondiente microinstrucción como salida. El contenido, la palabra direccionada en la ROM, permanece en las líneas de salida tanto tiempo como el valor de la dirección esté presente en las entradas. No es necesario emplear una señal de lectura/escritura como sucede si se usa una RAM. Cada pulso de reloj ejecuta las microoperaciones especificadas por la microinstrucción y, además, se transfiere una nueva dirección al CAR. En este caso, el CAR es el único componente del control que recibe los pulsos de reloj y almacena información. El generador de la siguiente dirección y la memoria de control son circuitos combinacionales, es decir, el contenido del CAR proporciona el estado de la unidad de control.

El control microprogramado es una técnica alternativa muy popular para realizar unidades de control tanto para sistemas programados como no programados. Sin embargo, según el sistema se hace más complejo, así como sus especificaciones, se ha incrementado la necesidad de secuencias paralelas concurrentes de acciones, y hace a la microprogramación menos atractiva a la hora de diseñar una unidad de control. Además, las ROM y RAM de gran capacidad son más lentas que la lógica combinacional equivalente. Para terminar, los HDLs y las herramientas de síntesis facilitan el diseño de las unidades de control complejas sin necesidad de utilizar un método de diseño programado. En general, el control microprogramado para diseñar unidades de control, particularmente dirigido a control de rutas de datos en CPUs, ha disminuido significativamente. Sin embargo, el control microprogramado ha surgido para utilizar arquitecturas de computadoras antiguas. Estas arquitecturas tienen un conjunto de instrucciones que no sigue los principios de las arquitecturas actuales. No obstante, dichas arquitecturas deben desarrollarse debido a las grandes inversiones en software para ellas. Además, los principios de las arquitecturas actuales se deben usar en aplicaciones para conseguir los requisitos perseguidos. El control de estos sistemas es jerárquico, con el control microprogramado situado en el nivel más alto de la jerarquía para la ejecución de instrucciones complejas, y a nivel inferior para llevar a cabo las instrucciones sencillas y las etapas de las instrucciones complejas muy rápidamente. El sentido de la microprogramación se trata en el Capítulo 12 en computadoras con conjunto de instrucciones complejo (CISC, del inglés *Complex Instruction Set Computer*).

Más información sobre el control microprogramado tradicional, extraído de pasadas ediciones de este texto, está disponible en un suplemento, Control Microprogramado, en la siguiente dirección de Internet: http://www.librosite.net/Mano.

8-8 RESUMEN DEL CAPÍTULO

En este capítulo se ha examinado la interacción entre las rutas de datos y las unidades de control y la diferencia entre sistemas programados y no programados. El algoritmo de máquinas de estados (ASM) es un medio para representar y especificar las funciones de control. Se ha usado un multiplicador binario para ilustrar la formulación de un diagrama ASM. Se han propuesto dos métodos para diseñar circuitos secuenciales: registro secuenciador con decodificador, y un flip-flop por estado, aparte del método de diseño básico del Capítulo 4. Se han mostrado modelos en VHDL y Verilog que describen la combinación de la ruta de datos y el control. Por último, se ha discutido brevemente el control microprogramado.

REFERENCIAS

1. MANO, M. M.: *Computer Engineering: Hardware Design:* Englewood Cliffs, NJ: Prentice Hall, 1988.

2. MANO, M. M.: *Digital Design*, 3rd Ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

3. *IEEE Standard VHDL Language Reference Manual*. (ANSI/IEEE Std 1076-1993; revision of IEEE Std 1076-1987). New York: The Institute of Electrical and Electronics Engineers, 1994.

4. SMITH, D. J.: *HDL Chip Design*. Madison, AL: Doone Publications, 1996.

5. *IEEE Standard Description Language Based on the Verilog(TM) Hardware Description Language* (IEEE Std 1364-1995). New York: The Institute of Electrical and Electronics Engineers, 1995.

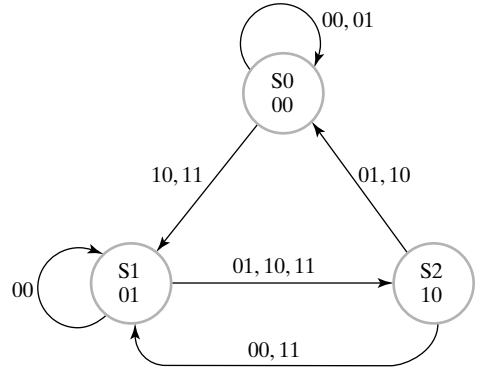
6. PALNITKAR, S.: *Verilog HDL: A Guide to Digital Design and Synthesis*. SunSoft Press (A Prentice Hall Title), 1996.

7. THOMAS, D. E., and P. R. MOORBY: *The Verilog Hardware Description Language* 4th ed. Boston: Kluwer Academic Publishers, 1998.

PROBLEMAS

El signo (+) indica problemas más avanzados y el asterisco (*) indica que una solución está disponible en la dirección de Internet: http://www.librosite.net/Mano.

8-1. *En la Figura 8-18 se muestra un diagrama de estados de un circuito secuencial. Dibuje el diagrama ASM correspondiente. Minimice su complejidad utilizando cajas de decisión vectorial y escalar. Las entradas del circuito son X_1 y X_2 y las salidas son Z_1 y Z_2.



□ **FIGURA 8-18**
Diagrama de estados para el Problema 8-1

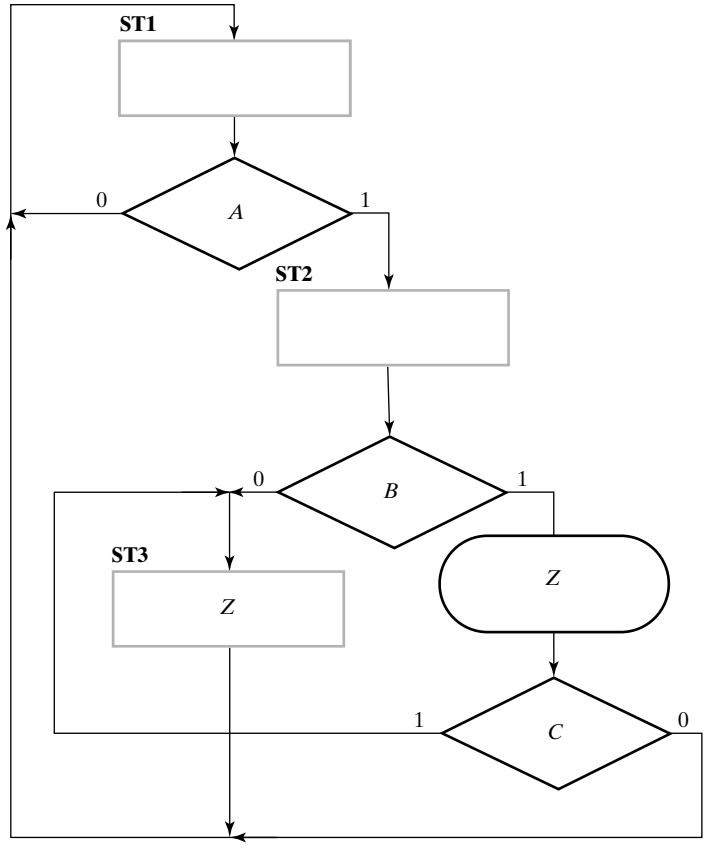
8-2. *Calcule la respuesta del diagrama ASM de la Figura 8-9 para la siguiente secuencia de entradas (suponga que el estado inicial es ST1):

A:	0	1	1	0	1	1	0	1
B:	1	1	0	1	0	1	0	1
C:	0	1	0	1	0	1	0	1

Estado: ST1

Z:

8-3. En la Figura 8-19 se muestra un diagrama ASM. Halle la tabla de estados para el circuito secuencial correspondiente.



□ **FIGURA 8-19**
Diagrama ASM para los Problemas 8-2 y 8-3

8-4. Encuentre el diagrama ASM correspondiente a la siguiente descripción: hay dos estados, *A* y *B*. Si en el estado *A* la entrada *X* es 1, el siguiente estado es *A*. Si en el estado *A* la entrada *X* es 0, el siguiente estado es *B*. Si en el estado *B* la entrada *Y* es 0, el siguiente estado es *B*. Si en el estado *B* la entrada *Y* es 1, el siguiente estado es *A*. La salida *Z* es igual a 1 mientras el circuito esté en el estado *B*.

8-5. *Proponga el diagrama ASM para un circuito que detecte un valor diferente de la señal de entrada *X* en dos flancos de subida consecutivos de reloj. Si *X* ha tenido valores diferentes en dos flancos de subida consecutivos de reloj la salida *Z* es igual a 1 en el siguiente ciclo de reloj, si no la salida *Z* es 0.

8-6. +Se quiere realizar el diagrama ASM de un circuito síncrono con reloj CK de una lavadora. El circuito tiene tres señales de entrada externas: START, FULL y EMPTY (las cuales están a uno, como mucho, durante un ciclo de reloj y se excluyen mutuamente), y las salidas externas son: HOT, COLD, DRAIN y TURN. La ruta de datos para el control se compone de un contador descendente que tiene tres entradas RESET, DEC y LOAD. El contador se decrementa síncronamente una vez cada minuto si DEC = 1, pero se puede

cargar o poner a cero síncronamente en cualquier ciclo de reloj CK. Tiene sólo una salida, ZERO, que vale 1 siempre y cuando el contenido del contador sea 0, en el resto de los casos vale 0.

Durante su funcionamiento, el circuito pasa a través de cuatro ciclos: WASH, SPIN, RINSE y SPIN, que se detallan a continuación:

WASH: suponga que el estado inicial del circuito al arrancar es IDLE. Si START es 1 durante un ciclo de reloj, HOT pasa a 1 y permanece a 1 hasta que FULL = 1, llenando la lavadora con agua caliente. A continuación, usando LOAD, el contador descendente se carga con un valor de un panel que indica cuántos minutos dura el ciclo de lavado. DEC y TURN pasan a 1 y la lavadora lava su contenido. Cuando ZERO es 1, el lavado se ha completado y TURN y DEC pasan a valer 0.

SPIN: luego, DRAIN pasa a 1, se vacía de agua la lavadora. Cuando EMPTY pasa a 0 el contador descendente se carga con el valor 7. DEC y TURN pasan a 1 y el agua restante se escurre de su contenido. Cuando ZERO pasa a 1, DRAIN, DEC y TURN vuelven a 0.

RINSE: posteriormente, COLD pasa a 1 y permanece a este valor hasta que FULL = 1, llenando la lavadora con agua fría para enjuagar. A continuación, empleando LOAD, el contador descendente se carga con el valor 10. DEC y TURN pasan a 1 y el agua enjuaga su contenido. Cuando ZERO pasa a 1, el enjuagado se ha completado y TURN y DEC pasan a 0.

SPIN: luego, DRAIN pasa a 1, sacando el agua del enjuague. Cuando EMPTY es 1 el contador descendente se carga con 8. DEC y TURN se ponen entonces a 1 y el agua restante del enjuague se escurre de su contenido. Cuando ZERO pasa a 1, DRAIN, DEC y TURN vuelven a valer 0 y el circuito pasa al estado IDLE.

(a) Halle el diagrama ASM del circuito de la lavadora.
(b) Modifique el diseño de la parte (a) suponiendo que hay dos entradas más, PAUSE y STOP. PAUSE hace que el circuito, incluyendo en contador, se detenga y ponga todas las salidas a 0. Cuando START se pulsa, la lavadora reanuda su funcionamiento en el punto donde se detuvo. Si se pulsa STOP, todas las salidas se ponen a 0 excepto DRAIN que se pone a 1. Si EMPTY se pone a 1, el estado del circuito vuelve a IDLE.

8-7. Proponga un diagrama ASM para el controlador de un semáforo que funciona de la siguiente manera: la señal de tiempos *T* es la entrada al controlador. *T* define el tiempo de la luz amarilla así como los cambios de rojo a verde. Las salidas a las señales se definen en la siguiente tabla:

Salida	Luz controlada
GN	Luz Verde, Semáforo Norte/Sur
YN	Luz Amarilla, Semáforo Norte/Sur
RN	Luz Roja, Semáforo Norte/Sur
GE	Luz Verde, Semáforo Este/Oeste
YE	Luz Amarilla, Semáforo Este/Oeste
RE	Luz Roja, Semáforo Este/Oeste

Mientras $T = 0$, la luz verde esta encendida para un semáforo y la luz roja para el otro. Con $T = 1$, la luz amarilla está encendida para el semáforo que fue previamente verde, y el semáforo que estuvo previamente en rojo permanece en rojo. Cuando T es 0, el semáforo que antes estuvo en amarillo pasa a rojo, y el semáforo que estuvo previamente en rojo pasa a ser verde. Este patrón de cambios alternados del color continúa. Suponemos que el controlador es síncrono con un reloj que cambia a mucha más frecuencia que la entrada *T*.

8-8. *Realice el diagrama ASM de la Figura 8-19 usando un flip-flop por estado.

8-9. *Realice el diagrama ASM de la Figura 8-19 mediante un registro de secuencia y un decodificador.

8-10. + Realice el diagrama ASM que se deriva del Problema 8-6(a) usando un flip-flop por estado.

8-11. *Multiplique los dos números binarios sin signo 100110 (multiplicando) y 110101 (multiplicador) usando el método manual y el método hardware.

8-12. Simule manualmente el proceso de multiplicar dos números binarios sin signo 1010 (multiplicando) y 1011 (multiplicador). Enumere el contenido de los registros *A*, *Q*, *P* y *C* y el estado del control. Utilice el sistema de la Figura 8-6 con *n* igual a 4 y con el control cableado de la Figura 8-12.

8-13. Determine el tiempo que necesita la operación de multiplicar en el sistema digital descrito en la Figura 8-6 y en la Figura 8-9. Suponga que el registro *Q* tiene *n* bits y el periodo de un ciclo de reloj es de f nanosegundos.

8-14. Compruebe que la multiplicación de dos números de *n* bits da un resultado de no más de $2n$ bits. Demuestre que esta condición implica que no puede haber *overflow* en el resultado final del circuito multiplicador de la Figura 8-6.

8-15. Considere el diagrama de bloques del multiplicador mostrado en la Figura 8-6. Suponga que el multiplicando y el multiplicador son de 16 bits.

(a) ¿En cuántos bits se puede expresar el resultado y cuándo está disponible?
(b) ¿De cuántos bits es el contador *P*, y qué número binario se debe cargar al comienzo?
(c) Diseñe el circuito combinacional que comprueba que el contador *P* es cero.

8-16. *Diseñe un sistema digital con tres registros de 16 bits *AR*, *BR* y *CR* y con datos de entrada de 16 bits, IN, para realizar las siguientes operaciones, suponiendo que se utiliza la representación de complemento a 2 e ignorando el *overflow*:

(a) Transferir dos números con signo de 16 bits a *AR* y *BR* en ciclos consecutivos de reloj después de que la señal *G* se ponga a 1.

(b) Si el número en *AR* es positivo y distinto de cero, multiplique el contenido de *BR* por dos y transfiera el resultado al registro *CR*.

(c) Si el número en *AR* es negativo, multiplique el contenido de *AR* por dos y transfiera el resultado al registro *CR*.

(d) Si el número en *AR* es cero, ponga a 0 el registro *CR*.

8-17. + Modifique el diseño del multiplicador de la Figura 8-6 y el diagrama ASM de la Figura 8-7 para llevar a cabo la multiplicación de números con signo en complemento a 2 usando el Algoritmo de Booth, que emplea un sumador-restador. La decisión de sumar o

restar o de no hacer nada se hace en función del bit menos significativo (LSB) del registro *Q* y del bit previo al LSB de registro *Q* antes de que *Q* sea desplazado a la derecha. De esta forma, se debe añadir un flip-flop para almacenar al bit previo al LSB del registro *Q*. El valor inicial del bit previo al LSB es 0. La siguiente tabla define las decisiones:

LSB de *Q*	LSB previo de *Q*	Acción
0	0	Dejar el producto parcial sin cambiar
0	1	Sumar el multiplicando al producto parcial
1	0	Restar el multiplicando del producto parcial
1	1	Dejar el producto parcial sin cambiar

8-18. +Diseñe un sistema digital que multiplique dos números binarios sin signo mediante el método de sumas repetidas. Por ejemplo, para multiplicar 5 por 4, el sistema digital suma el multiplicando cuatro veces: 5 + 5 + 5 + 5 = 20. Mantenga el multiplicando en el registro *BR*, el multiplicador en el registro *AR* y el producto en el registro *PR*. Un circuito sumador suma el contenido de *BR* a *PR*, y *AR* es un contador descendente. Un circuito que detecta 0, *Z*, comprueba cuándo *AR* es cero después de que se decrementa. Diseñe el control mediante el método de un flip-flop por estado.

8-19. *Escriba, compile y simule una descripción en VHDL para el diagrama ASM de la Figura 8-19. Use un conjunto de entradas en la simulación que haga pasar al diagrama ASM a través de todos sus caminos e incluya tanto el estado como la salida *Z* como resultados de la simulación. Corrija y vuelva simular su diseño si es necesario.

8-20. *Escriba, compile y simule una descripción en Verilog para el diagrama ASM de la Figura 8-19. Use los códigos 00 para el estado ST1, 01 para el estado ST2 y 10 para el estado ST3. Use un conjunto de entradas en la simulación que haga pasar al diagrama ASM a través de todos sus caminos e incluya tanto el estado como la salida *Z* como resultados de la simulación. Corrija y vuelva simular su diseño si es necesario.

8-21. Realice el diseño del Problema 8-5 usando Verilog en lugar de un diagrama ASM. Utilice para los estados los nombres S0, S1 y S2, ..., y los códigos que sean equivalentes a los enteros del nombre del estado. Compile y simule su diseño usando entradas en la simulación que valide completamente el diseño y que incluya tanto el estado como la salida *Z* como resultados de la simulación. Corrija y vuelva simular su diseño si es necesario.

8-22. +Realice el diseño del Problema 8-7 usando VHDL en lugar del diagrama ASM. Compile y simule su diseño haciendo funcionar el semáforo dos ciclos completos. Use un periodo real para *T* y un reloj lento. Ajuste el periodo de reloj, si es necesario para evitar simulaciones que lleven mucho tiempo.

8-23. +Realice el diseño del Problema 8-7 en Verilog en lugar del diagrama ASM. Compile y simule su diseño haciendo funcionar el semáforo dos ciclos completos. Use un periodo real para *T* y un reloj lento. Ajuste el periodo de reloj, si es necesario para evitar simulaciones que lleven mucho tiempo.

CAPÍTULO

9

MEMORIAS

La memoria es el componente más grande en una computadora digital y está presente en un gran porcentaje de los sistemas digitales. Las memorias de acceso aleatorio (RAM) almacenan datos temporalmente, y las memorias de sólo lectura (ROM) almacenan datos permanentemente. Una memoria ROM pertenece a un tipo de componentes llamados dispositivos lógicos programables (PLDs, del inglés *Programmable Logic Devices*) que utilizan la información almacenada para definir circuitos lógicos.

Nuestro estudio de las memorias RAM comienza viéndolas como un modelo con entradas, salidas y la temporización de sus señales. Usaremos, por tanto, modelos lógicos equivalentes para comprender el funcionamiento interno de las memorias RAM de los circuitos integrados. Se estudian las memorias RAM estáticas y dinámicas. También se estudian los distintos tipos de memorias RAM dinámicas usadas para el movimiento de datos a altas velocidades entre la CPU y la memoria. Finalmente juntaremos diversos chips de memoria RAM para construir la memoria de un sistema.

En algunos de los capítulos anteriores se utilizaron extensamente estos conceptos referentes a la computadora genérica al principio de Capítulo 1. En este capítulo, por primera vez vamos a ser más precisos y señalaremos los usos específicos de las memorias y sus componentes asociados. Empezando con el procesador, la caché interna es, básicamente, una memoria RAM muy rápida. Fuera de la CPU, la caché externa también es, básicamente, una memoria RAM muy rápida. El subsistema de memoria RAM, como su nombre indica, es un tipo de memoria. En la zona de entrada/salida, encontramos esencialmente memoria para almacenar información de la imagen de la pantalla en la tarjeta de vídeo. La memoria RAM aparece en la caché de disco en la tarjeta controladora del disco, acelerando los accesos a éste. Aparte del papel principal que tiene el subsistema de memoria RAM para almacenar datos y programas, encontramos memoria aplicada de varias formas en la mayoría de los subsistemas de una computadora genérica.

9-1 DEFINICIONES

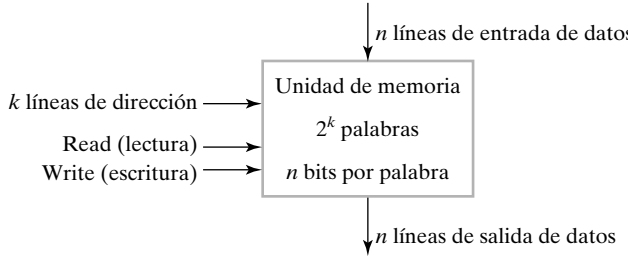
En los sistemas digitales, una memoria es una colección de celdas capaces de almacenar información binaria. Aparte de estas celdas, una memoria contiene circuitos electrónicos para almacenar y recuperar la información. Como se indicó en la explicación de la computadora genérica, la memoria se usa en diferentes partes de las computadoras modernas, proporcionando almacenamiento temporal o permanente para grandes cantidades de información binaria. Para que esta información sea procesada, se envía de la memoria al hardware de procesamiento, que está formado por registros y lógica combinacional. La información procesada se devuelve posteriormente a la misma memoria o a otra diferente. Los dispositivos de entrada y salida también interactúan con la memoria. La información de un dispositivo de entrada se coloca en la memoria, de forma que puede usarse en su procesado. La información procesada de salida se coloca en la memoria, y de allí se manda a un dispositivo de salida.

Se utilizan dos tipos de memoria en las diversas partes de una computadora: la *memoria de accesos aleatorio* (RAM, del inglés *random-access memory*) y la *memoria de sólo lectura* (ROM, del inglés *read-only memory*). La memoria RAM permite almacenar nueva información que estará disponible para su uso posteriormente. Al proceso de almacenamiento de nueva información en la memoria se le llama operación de *escritura* en la memoria. Al proceso de transferir la información almacenada en la memoria se le llama operación de *lectura* de la memoria. La memoria RAM puede realizar ambas operaciones, mientras que la memoria ROM, presentada en el Capítulo 3, sólo puede realizar operaciones de lectura. El tamaño de la memoria RAM puede variar entre cientos y millones de bits.

9-2 MEMORIA DE ACCESO ALEATORIO

Una memoria es una colección de celdas de almacenamiento binario junto con circuitos adicionales necesarios para transferir información de o desde un lugar determinado, con el mismo tiempo de acceso independientemente de dónde esté localizada, de aquí el nombre de *memoria de acceso aleatorio*. Por contra, la *memoria serie*, tal y como se produce en un disco magnético o en una unidad de cinta, necesita más o menos tiempo dependiendo de dónde este el dato deseado, puesto que depende de su localización física en el disco o en la cinta.

La información binaria se almacena en la memoria en grupos de bits, a cada grupo de bits se le llama *palabra* (del término inglés *word*). Una palabra es una entidad de bits que se mueve a dentro y a fuera de la memoria como una unidad, un grupo de unos y ceros que representan un número, una instrucción, uno o más caracteres alfanuméricos o cualquier otra información codificada. A un grupo de ocho bits se le llama byte. La mayoría de las computadora usan palabras que son múltiplo de ocho bits. Así, una palabra de 16 bits contiene dos bytes y una palabra de 32 bits está formada por cuatro bytes. La capacidad de una unidad de memoria se expresa como el número total de bytes que puede almacenar. La comunicación entre la memoria y su entorno se consigue mediante: líneas de entradas y de salidas de datos, líneas de selección de dirección y las líneas de control que especifican la dirección de la transferencia de la información. En la Figura 9-1 se muestra un diagrama de bloques de una memoria. Las n líneas de entrada proporcionan la información a almacenar en la memoria y las n líneas de salida aportan la información que sale de la memoria. Las k líneas de dirección especifican la palabra escogida de entre las muchas disponibles. Las dos entradas de control especifican el sentido de la transferencia deseada: la entrada de escritura (Write) provoca que el dato binario se transfiera dentro la memoria, y la entrada de lectura (Read) hace que el dato binario se transfiera a fuera de la memoria.



□ **FIGURA 9-1**
Diagrama de bloques de una memoria

La unidad de memoria se caracteriza por el número de palabras que contiene y por el número de bits en cada palabra. Las líneas de dirección seleccionan una palabra concreta. A cada palabra de la memoria se le asigna un número de identificación llamada *dirección* (en término inglés *address*). El rango de direcciones va desde 0 hasta $2^k - 1$, donde k es el número de líneas de dirección. La selección de una determinada palabra dentro de la memoria se hace aplicando la dirección en binario de k bits a las líneas de dirección. Un decodificador toma esta dirección y abre los caminos necesarios para seleccionar la palabra especificada. La memoria de una computadora puede tener muchos tamaños. Es habitual referirse al número de palabras (o bytes) mediante los prefijos K (kilo), M (mega) o G (giga). K es igual a 2^{10}, M es igual a 2^{20} y G es igual a 2^{30}. De esta forma, 64 K = 2^{16}, 2 M = 2^{21} y 4 G = 2^{32}.

Considere, por ejemplo, una memoria con una capacidad de 1 K palabras de 16 bits cada una. Puesto que 1 K = 1024 = 2^{10}, y 16 bits constituyen dos bytes, podemos decir que la memoria puede albergar 2048 o 2 K bytes. La Figura 9-2 muestra el posible contenido de las tres primeras y tres últimas palabras de la memoria de este tamaño. Cada palabra contiene 16 bits que se pueden dividir en dos bytes. Las palabras se reconocen por su dirección en decimal, desde 0 hasta 1023. Una dirección equivalente en binario tiene 10 bits. La primera dirección se especifica con 10 ceros y la última con 10 unos. Esto es así porque 1023 en binario es igual a 1111111111. Una palabra de memoria se selecciona mediante su dirección en binario. Cuando una palabra se lee o se escribe, la memoria funciona con todos los 16 bits como una única unidad.

Direcciones de memoria		
Binario	Decimal	Contenido de memoria
0000000000	0	10110101 01011100
0000000001	1	10101011 10001001
0000000010	2	00001101 01000110
	.	.
	.	.
	.	.
	.	.
	.	.
1111111101	1021	10011101 00010101
1111111110	1022	00001101 00011110
1111111111	1023	11011110 00100100

□ **FIGURA 9-2**
Contenido de una memoria de 1024 × 16

La memoria de 1 K × 16 de la figura tiene 10 bits en las direcciones y 16 bits en cada palabra. Si tuviésemos una memoria de 64 K × 10, sería necesario incluir 16 bits en las direcciones y cada palabra tendría 10 bits. El número de bits necesarios en las direcciones depende del número total de palabras que pueden ser almacenadas y es independiente del número de bits en cada palabra. El número de bits en la dirección de una palabra se determina mediante la relación $2^k \geqslant m$, donde m es el número total de palabras y k es el número mínimo de bits de direcciones que satisface la relación.

Operaciones de lectura y escritura

Las dos operaciones que puede efectuar una memoria de acceso aleatoria son la escritura (*write*) y la lectura (*read*). Una *escritura* es una transferencia al interior de la memoria de un nuevo dato para ser almacenado. Una lectura es una transferencia de una copia de una palabra almacenada al exterior de la memoria. Una señal de escritura (Write) especifica la operación de entrada, y una señal de lectura (Read) determina la operación de salida. Aceptando una de estas señales de control, los circuitos internos de la memoria permiten realizar la función deseada.

Los pasos que se deben realizan para realizar una operación de escritura son los siguientes:

1. Aplicar la dirección binaria de la palabra elegida a las líneas de dirección.
2. Aplicar los bits de datos que se deben almacenar en la memoria a las líneas de entrada de datos.
3. Activar la entrada de escritura (Write).

La unidad de memoria tomará los bits de las líneas de entrada de datos y los almacenará en la palabra especificada en las líneas de direcciones.

Los pasos que se deben seguir para realizar una operación de lectura son los siguientes:

1. Aplicar la dirección binaria de la palabra elegida a las líneas de dirección.
2. Activar la entrada de lectura (Read)

La memoria tomará los bits de la palabra que ha sido seleccionada por la dirección y los llevará a las líneas de salida de datos. El contenido de la palabra seleccionada no se cambiará con su lectura.

La memoria RAM se construye con circuitos integrados (chips) más circuitos lógicos adicionales. Habitualmente, los chips de memoria RAM tienen dos entradas de control para las operaciones de lectura y escritura en una configuración diferente a la descrita anteriormente. En lugar de tener dos entradas separadas para la lectura y la escritura, la mayoría de circuitos tiene, al menos, una entrada de selección de chip (*Chip Select*) que selecciona el chip que se va a leer o a escribir, y una entrada de lectura/escritura (Read/$\overline{\text{Write}}$) que determina la operación a realizar. Las operaciones de la memoria que resultan de esta configuración de las señales de control se muestran en la Tabla 9-1.

La selección del chip (en adelante *Chip Select*) se usa para habilitar uno o varios chips que forman la RAM y que contienen la palabra a la que se quiere acceder. Si el *Chip Select* se activa, la entrada Read/$\overline{\text{Write}}$ determina la operación a realizar. Mientras la señal de *Chip Select* accede a los chips, también se proporciona una señal que accede a toda la memoria. Llamaremos a esta señal Habilitación de Memoria (*Memory Enable*).

□ **TABLA 9-1**
Entradas de control de un chip de memoria

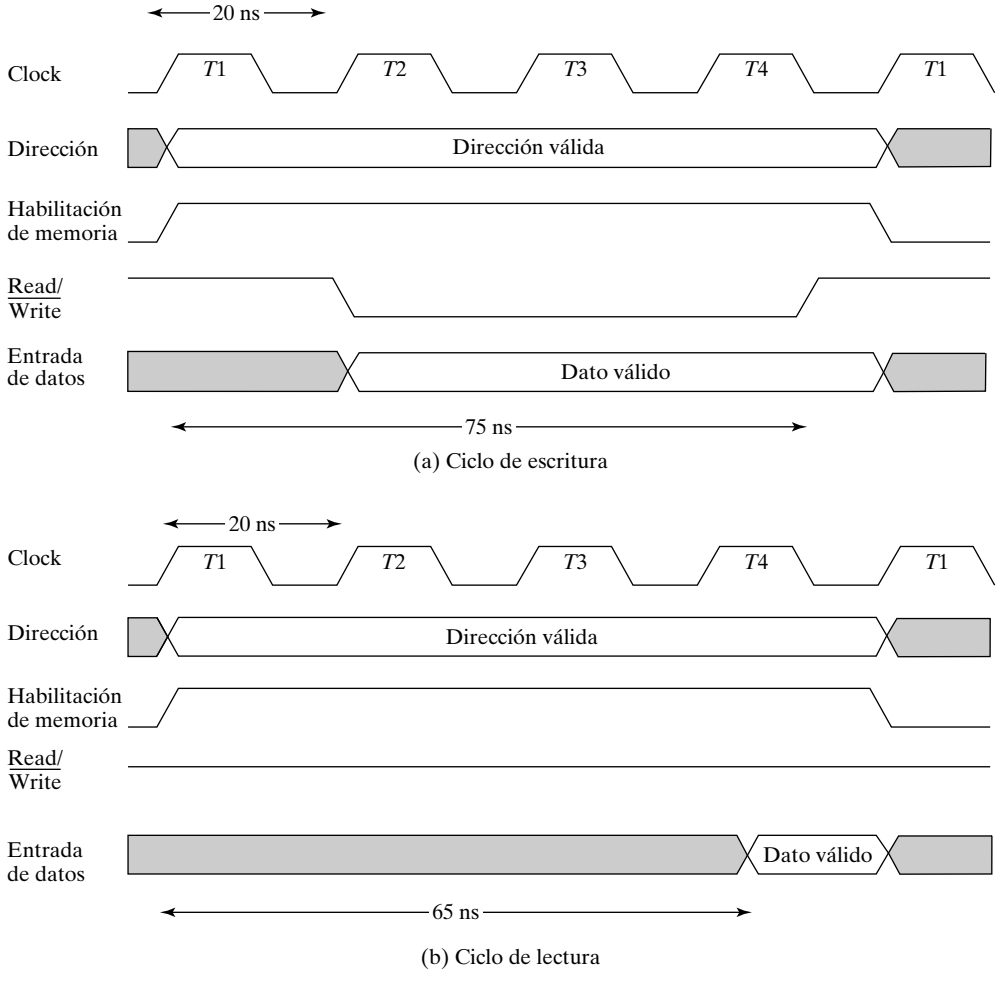
Chip select *CS*	*Read*/$\overline{\textit{Write}}$ $R/\overline{W}$	Operación de la memoria
0	×	Ninguna
1	0	Escriba la palabra seleccionada
1	1	Lee de la palabra seleccionada

Temporización de las formas de onda

El funcionamiento de la unidad de memoria se controla mediante un dispositivo externo, como por ejemplo, una CPU. La CPU está sincronizada con su señal de reloj. Sin embargo, la memoria no emplea este reloj. En cambio, sus operaciones de lectura y escritura se temporizan mediante cambios en sus entradas de control. El *tiempo de acceso* de una operación de lectura es el tiempo máximo que transcurre desde la aplicación de la dirección hasta que aparece la información en la salida de datos. De forma similar, el *tiempo del ciclo de escritura* es el tiempo máximo que transcurre desde que se pone la dirección hasta completar todas las operaciones internas que necesita la memoria para almacenar una palabra. Las escrituras en la memoria se pueden llevar a cabo una detrás de otra en intervalos de tiempo. La CPU debe proporcionar a la memoria señales de control de tal forma que se sincronizan sus propias operaciones internas, sincronizadas con el reloj, con las operaciones de lectura y escritura de la memoria. Esto significa que el tiempo de acceso y el tiempo del ciclo de escritura de la memoria deben estar relacionados con la CPU con un periodo igual a un número fijo de ciclos de reloj de la CPU.

Supongamos, como ejemplo, que una CPU funciona con un reloj de 50 MHz de frecuencia, que tiene un periodo de reloj de 20 ns (1 ns = 10^{-9} s). Supongamos ahora que la CPU se comunica con una memoria con un tiempo de acceso de 65 ns y un tiempo ciclo de escritura de 75 ns. El número de ciclos de reloj necesarios para una petición de memoria es un valor entero mayor o igual que el valor más grande del tiempo de acceso y del tiempo del ciclo de escritura, dividido por el periodo del reloj. Como el periodo del reloj es de 20 ns y el tiempo mayor entre el tiempo de acceso y el tiempo del ciclo de escritura es 75 ns, será necesario emplear, al menos, cuatro ciclo de reloj para cada petición a la memoria.

La temporización del ciclo de memoria se muestra en la Figura 9-3, para una CPU funcionando a 50 MHz y una memoria con un ciclo de escritura de 75 ns y un tiempo de acceso de 65 ns. El ciclo de lectura de la parte (a) muestra 4 pulsos *T*1, *T*2, *T*3 y *T*4 con un periodo de 20 ns. En una operación de escritura, la CPU debe proporcionar la dirección y el dato de entrada a la memoria. Se aplica la dirección y la habilitación de memoria se cambia a 1 en el flanco de subida del pulso *T*1. El dato, que es necesario algo más tarde, se aplica en el flanco de subida de *T*2. Las dos líneas que se cruzan una con otra en las forma de ondas de las direcciones y de los datos indica un posible cambio en el valor de estos buses. Las zonas sombreadas representan valores sin especificar. Un cambio de la señal Read/$\overline{\text{Write}}$ a 0 en el flanco positivo de *T*2 indica la operación de escritura. Para evitar la destrucción de los datos, en otras palabras de la memoria, es importante que este cambio ocurra después de que las señales en el bus de direcciones estén estables y con el valor de la dirección deseada. De lo contrario, una o más palabras pueden ser direccionadas momentáneamente y ser sobrescritas con diferentes datos. La señal debe



□ **FIGURA 9-3**
Formas de onda del ciclo de memoria

la habilitación de memoria para poder completar la operación de escritura. Finalmente, la dirección y el dato deben permanecer estables un poco después de que la señal *Read*/*Write* cambie a 1 de nuevo para evitar destruir los datos en otras palabras de la memoria. Después del cuarto pulso de reloj, la operación de escritura ha terminado con 5 ns de sobra, y la CPU puede poner la dirección y las señales para hacer otra petición de memoria en el siguiente pulso *T*1.

El ciclo de lectura de la Figura 9-3(b) tiene una dirección para la memoria que proporciona la CPU. La CPU pone la dirección, cambia la habilitación de memoria a 1 y la señal Read/Write a 1 para seleccionar la operación de lectura, todo esto en el flanco de subida de *T*1. La memoria coloca el dato de la palabra seleccionada, mediante la dirección en el bus de salida de datos, dentro de los 65 ns a partir de que se pone la dirección y se activa la habilitación de la memoria. Luego, la CPU transfiere el dato a uno de sus registros internos en el flanco de subida del siguiente pulso *T*1, en el que también se puede cambiar la dirección y las señales de control para la siguiente petición de memoria.

Características de las memorias

Las memorias en los circuitos integrados pueden ser estáticas o dinámicas. Las Memorias *Estáticas* (SRAM, del inglés *Static RAM*) están formadas con *latches* internos que almacenan la información binaria. La información permanece almacenada correctamente mientras que la memoria RAM esté alimentada. Las Memorias *Dinámicas* (DRAM, del inglés *Dynamic RAM*) almacenan la información binaria mediante cargas eléctricas en condensadores. Estos condensadores se fabrican dentro del chip utilizando transistores MOS de canal n. La carga almacenada en el condensador tiende a descargarse con el tiempo, por lo que los condensadores deben ser recargados periódicamente mediante el *refresco* de la memoria DRAM. Esto se hace cíclicamente en todas las palabras cada pocos milisegundos, leyendo y rescribiéndolos para restablecer la carga perdida. Las memorias DRAM de los chips son de bajo consumo y de gran capacidad de almacenamiento pero las memorias SRAM son más fáciles de usar y tienen unos ciclos de lectura y escritura más cortos, y además, no necesitan un ciclo de refresco.

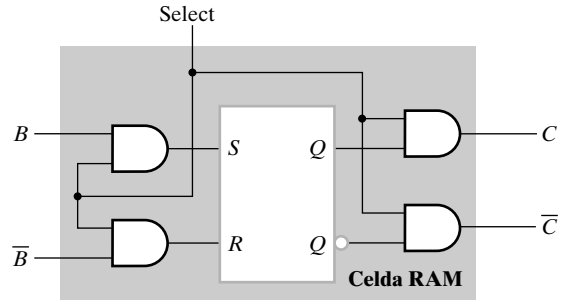
Las unidades de memorias que pierden la información cuando se apaga la alimentación se dicen que son *volátiles*. Las RAM de los circuitos integrados, tanto estáticas como dinámicas, pertenecen a esta categoría puesto que sus celdas necesitan una fuente de alimentación externa para mantener la información almacenada. Por el contrario, una *memoria no volátil*, como son los discos magnéticos, mantienen su información después de desconectar la alimentación. Esto es así porque los datos almacenados en soporte magnético se representan mediante la orientación de la magnetización, que no se pierde al desconectar la fuente de alimentación. Las memorias ROM son también memorias no volátiles, como se vio en la Sección 3-9.

9-3 MEMORIAS INTEGRADAS SRAM

Como se indicó anteriormente, las memorias están compuestas por circuitos integrados con memoria RAM más lógica adicional. Primero veremos la estructura interna de una memoria integrada RAM y luego estudiaremos las combinaciones de los chips de memoria RAM y la lógica adicional usada para construir una memoria. La estructura interna de un chip de memoria RAM de m palabras de n bits por palabra está compuesta de un array de $m \cdot n$ celdas de almacenamiento binario y una circuitería asociada. El circuito se construye con un decodificador para seleccionar la palabra que se va a leer o a escribir, circuitos de lectura, circuitos de escritura y lógica de salida. La celda de una memoria RAM es la célula de almacenamiento básica usada en un chip de una memoria RAM, que se diseña típicamente como un circuito electrónico en lugar de un circuito lógico. Sin embargo, es posible y conveniente modelar la memoria RAM de un chip usando un modelo lógico.

Usaremos un chip de memoria RAM estática como base para nuestra explicación. Primero presentamos la lógica de una célula RAM que almacena un solo bit y posteriormente usamos la célula jerárquicamente para describir un chip de memoria RAM. La Figura 9-4 muestra el modelo lógico de una celda de memoria RAM. La parte de almacenamiento de la celda se modela con un latch *SR*. Las entradas del latch se habilitan con la señal de selección, Select. Para Select igual a 0, el valor almacenado se retiene. Para Select igual a 1, el valor almacenado se determina mediante los valores de B y $\overline{B}$. Las salidas del latch se habilitan con Select, usando una puerta AND, para generar las salidas de la celda C y $\overline{C}$. Para Select igual 0, ambas salidas son 0 y para Select igual a 1, C tiene el valor almacenado y $\overline{C}$ tiene su complemento.

Para obtener un diagrama interconectamos un conjunto de celdas de la memoria RAM y los circuitos de lectura y escritura para construir una tira de un bit de una memoria RAM que



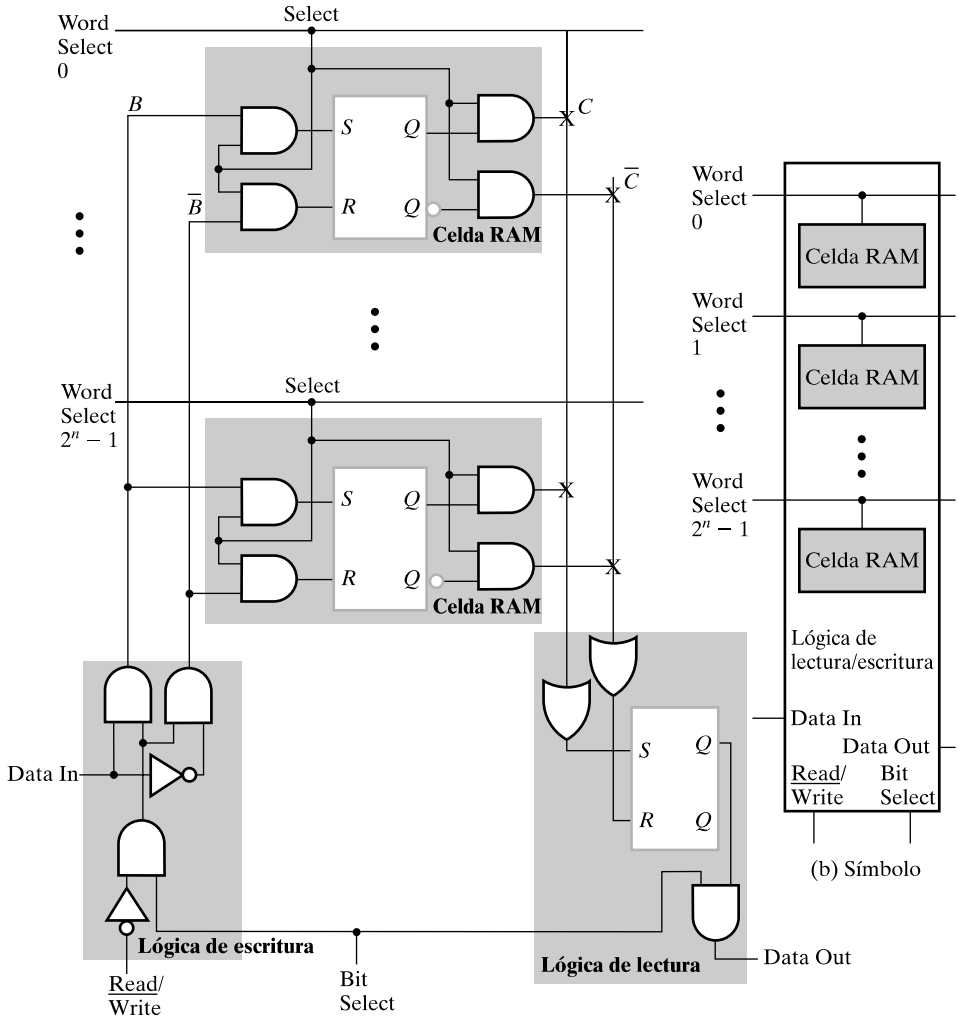
□ **FIGURA 9-4**
Celda de una memoria RAM estática

contiene todos los circuitos asociados con una posición de un bit de un conjunto de palabras de la memoria RAM. En la Figura 9-5 se muestra el diagrama lógico de una tira de un bit de memoria RAM. La parte del modelo que representa a cada celda de la memoria RAM se resalta en azul. La carga del latch de la célula se controla ahora con una entrada de selección de palabra, *Word Select*. Si ésta es 0, *S* y *R* son 0 y el contenido del latch de la célula permanece sin variar. Si la entrada *Word Select* es 1, entonces el valor a cargar en el latch se controla con las dos señales *B* y que parten de la lógica de escritura. Para que cualquiera de estas señales sea 1 y puedan cambiar el valor almacenado, Read/$\overline{\text{Write}}$ debe ser 0 y la señal *Bit Select* debe ser 1. Entonces, el dato de entrada, *Data_In*, y su complemento se aplica a *B* y , respectivamente, para poner a uno o a cero el latch de la celda de la memoria RAM seleccionada. Si *Data_In* es 1, el *latch* se pone a 1 y si es 0 se pone a 0, completando la operación de escritura.

Solamente se escribe una palabra cada vez. Es decir, sólo una línea *Word Select* es 1 y todas las demás son 0. Así, sólo la celda de la memoria RAM conectada a *B* y $\overline{B}$ se escribe. La señal *Word Select* también controla la lectura de las celdas de la memoria RAM usando la lógica compartida de la escritura. Si *Word Select* es 0, el valor almacenado en el *latch SR* se bloquea mediante las puertas AND para que no alcance a las dos puertas OR de la lógica de lectura. Pero si *Word Select* es 1, el valor almacenado pasa a través de las puertas OR y se captura en el *latch SR* de la lógica de lectura. Si la señal *Bit Select* también es 1, el valor capturado también aparece en la línea *Data Out* de la tira de un bit de la memoria RAM. Véase que, para el diseño de esta lógica de lectura en particular, la lectura ocurre independientemente del valor de la señal Read/$\overline{\text{Write}}$.

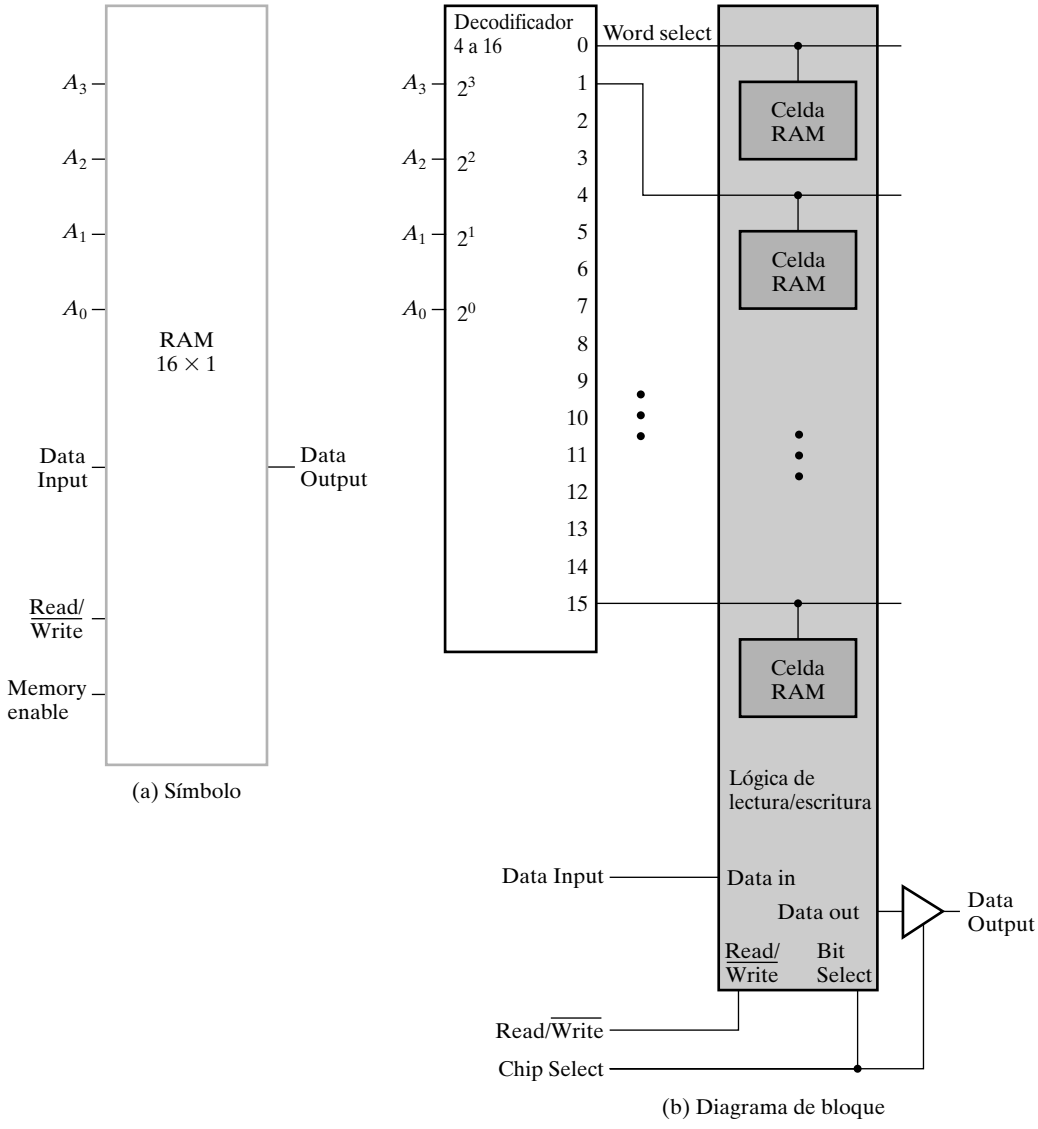
El símbolo de la tira de un bit de una memoria RAM, que se da en la Figura 9-5(b), se utiliza para representar la estructura interna de un chip de memoria RAM. Cada línea de selección se extiende más allá de la tira de un bit de forma que cuando varias tiras de un bit se colocan una junto a otra, se conectan las correspondientes líneas de selección. Las otras señales de la parte baja del símbolo se pueden conectar de diversas formas dependiendo de la estructura del chip de memoria RAM.

En la Figura 9-6 se muestra el símbolo y el diagrama de bloques de un chip de una memoria RAM de 16 × 1. Ambos tienen 4 entradas de direcciones para las 16 palabras de un bit almacenadas en la memoria RAM. La habilitación del chip, Chip Select, a nivel del chip se corresponde con la habilitación de memoria, *Memory Enable*, a nivel de la memoria RAM formada por varios chips. La estructura interna de la RAM está compuesta por una tira de un bit de memoria RAM que tiene 16 celdas. Como hay 16 líneas de selección de palabra que controlar, de forma que una y solo una tienen un 1 lógico en un instante dado, se utiliza un decodificador de 4 a 16 líneas para descodificar los cuatro bits de dirección a los 16 bits de la palabra de selección.



□ **FIGURA 9-5**
Modelo de una tira de un bit de la memoria RAM

La única lógica adicional en la figura es un triángulo con una entrada normal, una salida normal y una segunda entrada debajo del símbolo. Este símbolo es un buffer tri*estado* que permite la construcción de un multiplexor con un número arbitrario de entradas. Las salidas triestado se conectan juntas y se controlan adecuadamente usando las entradas de Chip Select. Con el uso de los buffers tri-estado en las salidas de la memoria RAM, dichas salidas pueden unirse para sacar la palabra del chip cuando éste se lee de las líneas de salidas conectadas a las salidas de la RAM. Las señales de habilitación anteriormente comentadas se corresponden con las entradas de Chip Select de los chips de la memoria RAM. Para leer una palabra un chip de memoria RAM concreto, el valor de Chip Select para ese chip debe ser 1 y para los chips restantes, que están conectados a las mismas líneas de salida, el Chip Select deben ser 0. Estas combinaciones que contienen un único 1 se pueden obtener de un decodificador.



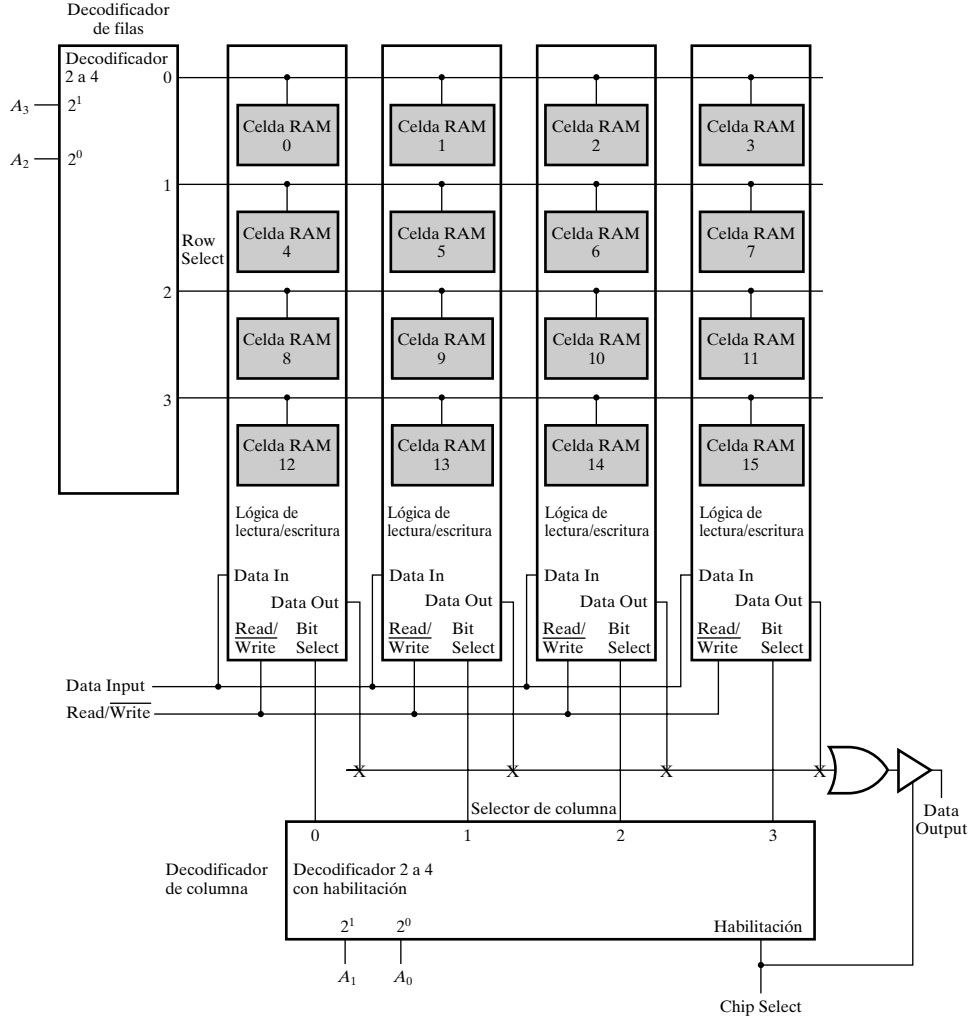
□ **FIGURA 9-6**
Chip de memoria RAM de 16 palabras de 1 bit.

Selección combinada

Dentro de un chip de memoria RAM, el decodificador de k entradas y 2^k salidas necesita 2^k puertas AND con k entradas por puerta si se utiliza un método de diseño directo. Además, si el número de palabras es grande, y todos los bits correspondientes a una posición de una palabra está en una única tira de un bit de una memoria RAM, el número de celdas RAM compartiendo los circuitos de lectura y escritura también es grande. Las características eléctricas resultantes de estas dos situaciones hacen que los tiempos de acceso de lectura y escritura de la memoria RAM sean largos, lo que cual es indeseable.

El número total de puertas del decodificador, el número de entradas por puerta y el número de celdas de memoria RAM por tira de un bit pueden reducirse empleando dos decodificadores empleando un método de *selección combinada*. Una posible configuración es utilizar dos decodificadores de $k/2$ entradas en lugar de uno de k entradas. Un decodificador controla las líneas de selección de palabra y el otro controla las líneas de selección de bit. El resultado es un método de selección con una matriz bidimensional. Si el chip de memoria RAM tiene m palabras de 1 bit por palabra, el método selecciona la célula de la memoria que está en la intersección de la fila de selección de palabra y la columna de selección de bit. Como *Word Select* no selecciona estrictamente palabras, su nombre se cambia a Row Select (selección de fila). A las salidas del decodificador añadido, que selecciona una o más tiras de bits se les llamará Column *Select* (selección de columna).

En la Figura 9-7 se muestra el método de selección combinada para un chip de memoria RAM. El chip esta formado por cuatro tiras de bits con cuatro bits en cada una y tiene un

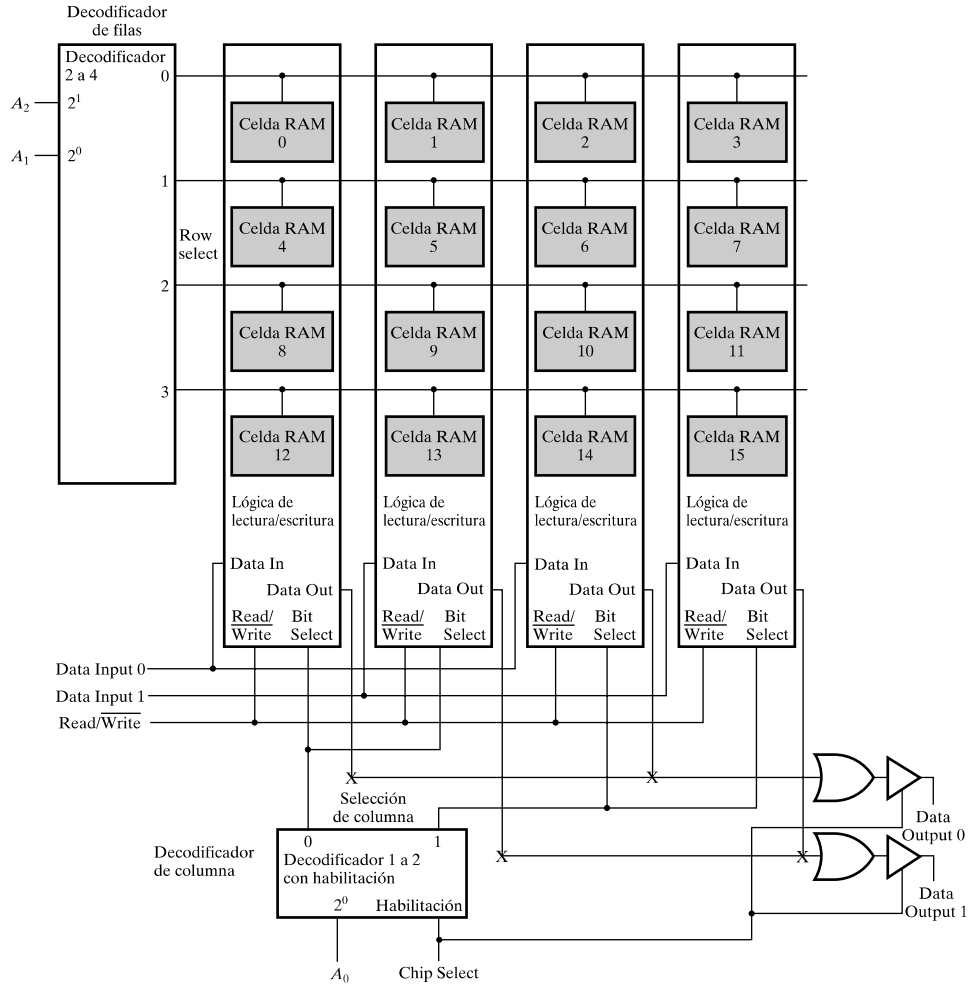


□ **FIGURA 9-7**
Diagrama de una memoria RAM de 16 × 1 utilizando un array de 4 × 4 celdas de memoria RAM

total de 16 celdas de memoria RAM en un array de dos dimensiones. Los dos bits más significativos del bus de direcciones van al decodificador de 2 a 4 líneas de las filas para seleccionar una de las cuatro filas del array. Las dos líneas menos significativas del bus de direcciones van al decodificador de 2 a 4 líneas de las columnas para seleccionar una de las cuatro columnas (tiras de un bit) del array. El decodificador de las columnas se habilita con la entrada de Chip Select. Cuando el Chip Select es 0, todas las salidas del decodificador están a 0 y no se selecciona ninguna de las celdas. Esto evita la escritura en cualquier celda del array de la memoria RAM. Cuando Chip Select es 1, se accede a un solo bit de la memoria. Por ejemplo, para la dirección 1001, los dos primeros bits de direcciones se decodifican para seleccionar la fila 10 (2_{10}) del array de celdas de la memoria. Los otros dos bits de direcciones se decodifican para seleccionar la columna 01 (1_{10}) del array. La celda de la memoria a la que se accede, en la fila 2 y columna 1 del array, es la celda 9 (10_2 01_2). Una vez seleccionada la fila y la columna, la entrada Read/$\overline{\text{Write}}$ determina la operación a realizar en la memoria. Durante la operación de lectura (Read/$\overline{\text{Write}}$ = 1), el bit seleccionado de la columna seleccionada pasa por la puerta OR al buffer triestado. Nótese que la puerta se dibuja de acuerdo con el array lógico presentado en la Figura 3-22. Como el buffer se habilita con la señal Chip Select, el valor leido aparece en la salida de datos, Data Output. En la operación de escritura (Read/$\overline{\text{Write}}$ = 0), el bit disponible en la línea de entrada de datos, Data Input, el bit disponible en la línea de entrada de datos, Data Input, se transfiere a la celda seleccionada de la memoria. El resto de las celdas no seleccionadas de la memoria están deshabilitadas y sus valores almacenados permanecen sin cambiar.

El mismo array de celdas se usa en la Figura 9-8 para construir una memoria RAM de 8 × 2 (ocho palabras de 2 bits). La decodificación de las filas no cambia con respecto de la Figura 9-7; los únicos cambios están en las columnas y en la lógica de salida. Al tener solo tres bits de direcciones y utilizar dos de ellos para el decodificador de filas, el decodificador de las columnas utiliza como entradas al bit de direcciones restante y la entrada Chip Select para generar dos líneas de selección de columna. Puesto que se quiere leer o escribir dos bits simultáneamente, las líneas de selección de columna van conectadas a los pares de tiras de un bit adyacentes. Las dos líneas de entrada, Data Input 0 y Data Input 1, van cada una a diferentes bits de cada uno de los pares. Finalmente, los bits correspondientes de cada par comparten la salida de las puertas OR y de los buffers triestado, dando lugar a las líneas de salida Data Output 0 y Data Output 1. La forma de operar de esta estructura se puede ilustrar mediante la aplicación de la dirección 3 (011_2). Los primeros dos bits de la dirección, 01, seleccionan la fila 1 del array. El último bit, 1, selecciona la columna 1, compuesta por las tiras 2 (10_2) y 3 (11_2) de un bit. De esta forma, la palabra a escribir o a leer está en las celdas 6 y 7 de la memoria (011 0_2 y 011 1_2), que contienen a los bits 0 y 1, respectivamente, de la palabra 3.

Podemos demostrar el ahorro de la selección combinada considerando una memoria RAM estática de un tamaño real, 32 K × 8. Esta memoria contiene un total de 256 K bits. Haciendo el número de columnas del array igual al de filas, calculamos la raíz cuadrada de 256 K, que da un resultado de $512 = 2^9$. De esta forma, los primeros nueve bits de dirección se conectan al decodificador de filas y los seis restantes al decodificador de columnas. Sin selección combinada, el único decodificador tendría 15 entradas y 32.768 salidas. Con selección combinada, hay un decodificador de 9 a 512 líneas y uno de 6 a 64 líneas. El número de puertas en el diseño con un solo decodificador sería de 32 800. En el caso de los dos decodificadores combinados, el número de puertas es 608, reduciéndose el número de puertas por un factor mayor que 50. Además, aunque parezca que hay 64 veces tantos circuitos de lectura/escritura, la selección de columna se puede hacer entre las celdas de la memoria RAM y los circuitos de lectura/escritura, así que sólo se necesitan los ocho circuitos originales. Debido al reducido número de celdas de memoria conectadas a cada circuito de lectura/escritura, los tiempos de acceso del chip también se mejoran.

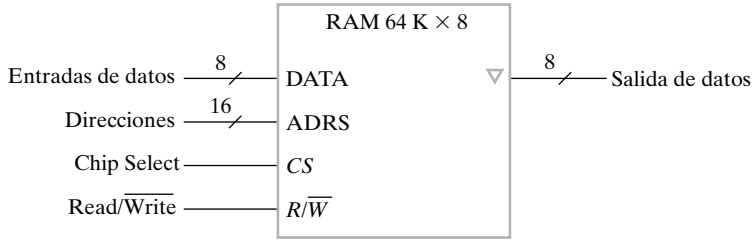


□ **FIGURA 9-8**
Diagrama de bloques de una memoria RAM de 8 × 21 utilizando un array de 4 × 4 celdas de memoria RAM

9-4 ARRAY DE CIRCUITOS INTEGRADOS DE MEMORIA SRAM

Los circuitos integrados de memoria RAM están disponibles en diversos tamaños. Si la unidad de memoria necesaria en una aplicación determinada es mayor que la capacidad de un chip, es necesario combinar un número de circuitos integrados en un array para construir la memoria con el tamaño requerido. La capacidad de la memoria depende de dos parámetros: el número de palabras y el número de bits por palabra. Un incremento en el número de palabras obliga a que incrementemos la longitud de la dirección. Cada bit añadido a la longitud de la dirección dobla el número de palabras en la memoria. Un incremento en el número de bits por palabra obliga a que incrementemos el número de líneas de entrada y salida de datos pero la longitud de la dirección permanece constante.

Para mostrar un array de circuitos integrados de memoria RAM vamos a utilizar un chip de memoria RAM, usando una representación condensada sus entradas y salidas, según se muestra en la Figura 9-9. La capacidad del chip es de 64 K palabras de 8 bits cada una. El integrado



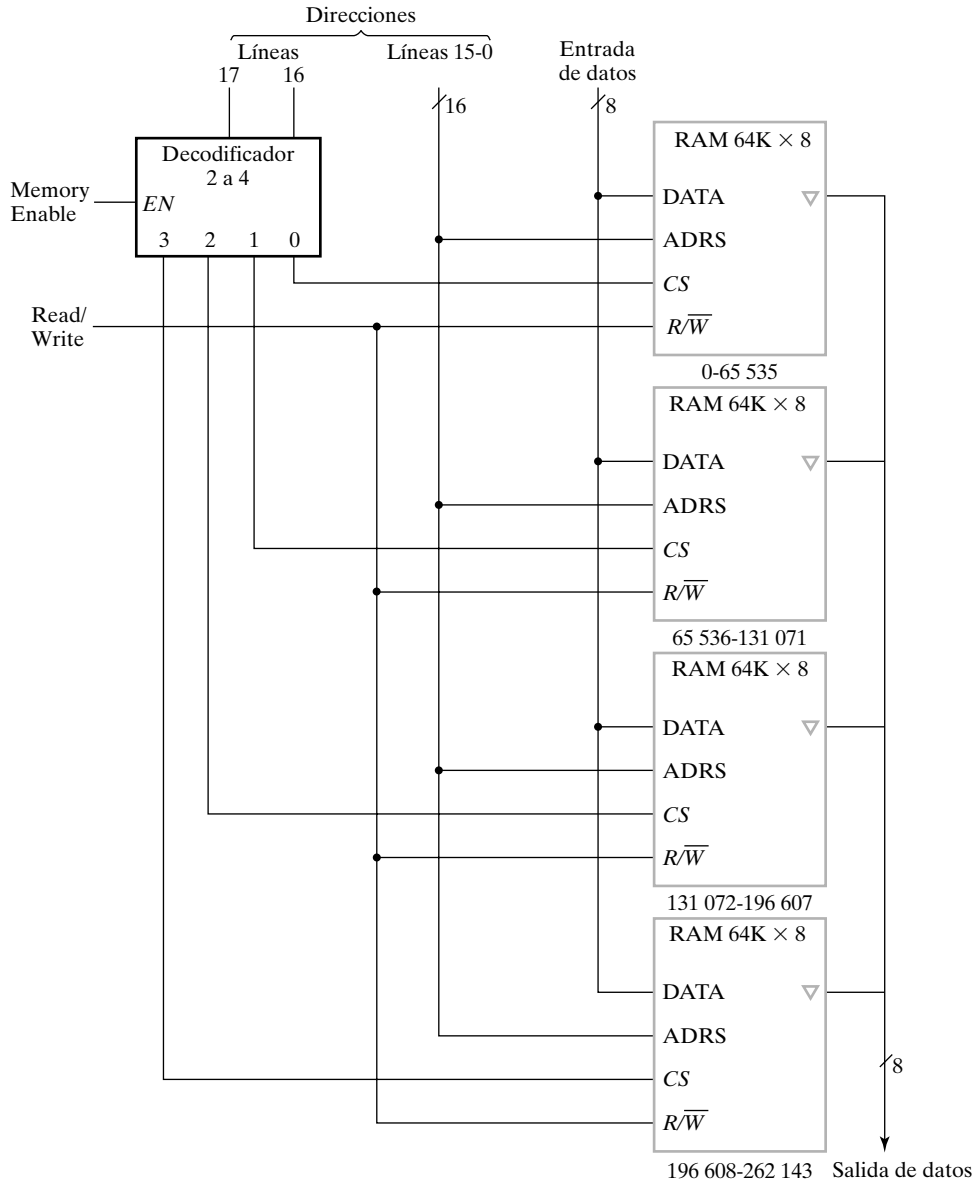
□ **FIGURA 9-9**
Símbolo de una memoria RAM de 64 × 8

necesita, por tanto, 16 líneas de direcciones, 8 líneas para la entrada y 8 líneas para la salida. En lugar de usar 16 líneas para las direcciones y 8 para la entrada y la salida, cada conjunto de líneas se mostrará en el diagrama de bloques como una línea simple. Cada línea está cruzada por una línea inclinada con un número que indica el número de líneas representadas en cada bus. La entrada *CS* (Chip Select) selecciona a un chip concreto de la memoria RAM y la entrada $R/\overline{W}$ (Read/$\overline{\text{Write}}$) especifica la operación de lectura o escritura cuando el chip ha sido seleccionado. El triángulo pequeño a la salida es el símbolo estándar para representar las salidas tri-estado. La entrada *CS* de la memoria RAM controla el comportamiento de las líneas de salida. Cuando $CS = 0$, el chip no ha sido seleccionado y todas las líneas de salida están en estado de alta impedancia. Cuando $CS = 1$, las líneas de salida de datos llevan los ocho bits de la palabra seleccionada.

Suponga que queremos incrementar el número de palabras en la memoria usando dos o más chips de memoria RAM. Puesto que por cada bit que se añade a la dirección se dobla el número binario que se puede formar, la forma natural de incrementar el número de palabras es por un factor de dos. Por ejemplo, dos chips de memoria doblarán el número de palabras y se añadirá un bit más para componer la dirección. Cuatro chips de memoria multiplican el número de palabras por cuatro y se añaden dos bits más para componer la dirección.

Considere la posibilidad de construir una memoria RAM de 256 K × 8 con cuatro chips de memoria RAM de 64 K × 8, como se muestra en la Figura 9-10. Las ocho líneas de datos llegan a todos los chips. Las salidas tri-estado se pueden conectar para formar un bus de salida de datos común. Este tipo de conexión de salida sólo es posible con salidas tri-estado. En cualquier instante, solamente se activará una entrada de selección de un chip, mientras que los restantes estarán deshabilitados. Las ocho salidas del chip seleccionado tendrán unos y ceros y las salidas de los otros tres estarán en estado de alta impedancia, presentándose sólo como circuitos abiertos a las señales de salida del circuito seleccionado.

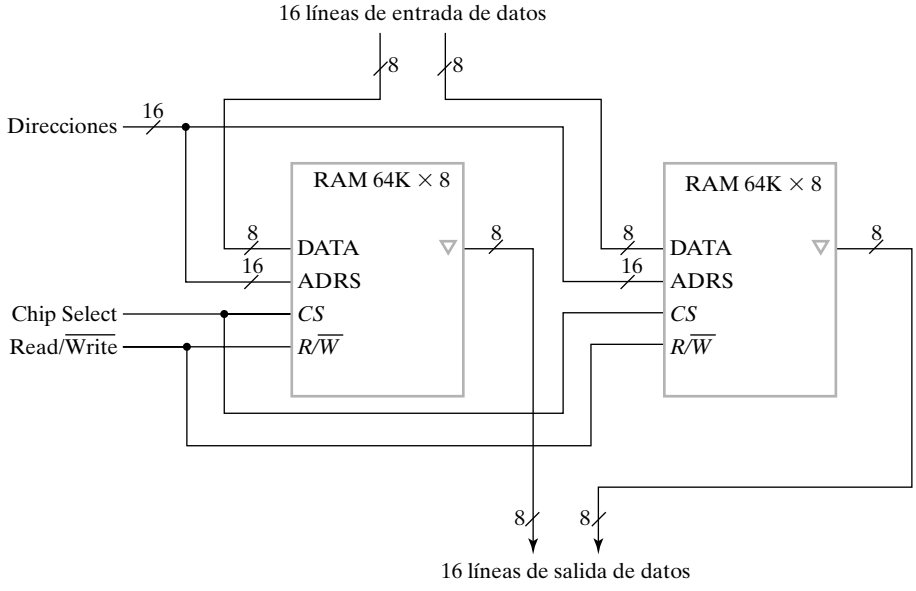
La memoria de 256 K palabras necesita un bus de direcciones de 18 bits. Los 16 bits menos significativos se conectan a las entradas de direcciones de los cuatro chips. Los dos bits más significativos se llevan a las entradas de un decodificador de 2 a 4 líneas. Las cuatro salidas del decodificador se aplican a las entradas *CS* de los cuatro chips. La memoria se deshabilita cuando la entrada *EN* del decodificador, *Memory enable*, es igual a 0. Las cuatro salidas del decodificador son 0 y ningún chip está siendo seleccionado. Cuando el decodificador está habilitado, los bits de direcciones 17 y 16 determinan cuál de los cuatro chips ha sido seleccionado. Si estos bits son igual a 00, el chip de memoria seleccionado es el primero. El resto de los 16 bits de direcciones seleccionan entonces una palabra dentro del chip en el rango de 0 a 65 535. Las siguientes 65 535 palabras se seleccionan del segundo chip de memoria con una dirección de 18 bits que empieza con 01 seguido de los 16 bits restantes de las líneas comunes del bus de direcciones. El rango de direcciones para cada chip se enumera en decimal debajo de su símbolo.



□ **FIGURA 9-10**
Diagrama de bloques de una memoria RAM de 256 K × 8

También es posible combinar dos chips para formar una memoria compuesta que contenga el mismo número de palabras pero con el doble de bits en cada palabra. En la Figura 9-11 se muestra la interconexión de dos chips para formar una memoria de 64 K × 16. Las 16 líneas de entrada y de salida se dividen entre los chips. Ambos reciben los 16 bits de direcciones y las entradas comunes de control *CS* y $R/\overline{W}$.

Las dos técnicas que se acaban de describir pueden combinarse para montar un array de chips idénticos para formar una memoria de gran capacidad. La memoria compuesta tendrá un número de bits por palabra que será múltiplo del número de bits por palabra de cada chip.



□ **FIGURA 9-11**
Diagrama de bloques de una memoria RAM de 64 K × 16

El número total de palabras se incrementará por un factor de dos veces la capacidad de palabras de un chip. Se necesita utilizar un decodificador externo para seleccionar a los chips individualmente según los bits adicionales de la memoria compuesta. Para reducir el número de pines del encapsulado del chip, muchos circuitos integrados tienen terminales comunes para la entrada y salida de datos. En este caso se dice que los terminales son bidireccionales, que quiere decir que para la operación de lectura éstos actúan como salidas y para la operación de escritura funcionan como entradas. Las líneas bidireccionales se construyen con buffers triestado, que ya se explicaron en la Sección 2-8. El uso de señales bidireccionales necesitan el control de los buffers triestado mediante las señales Chip Select y Read/$\overline{\text{Write}}$.

9–5 CIRCUITOS INTEGRADOS DE MEMORIA DRAM

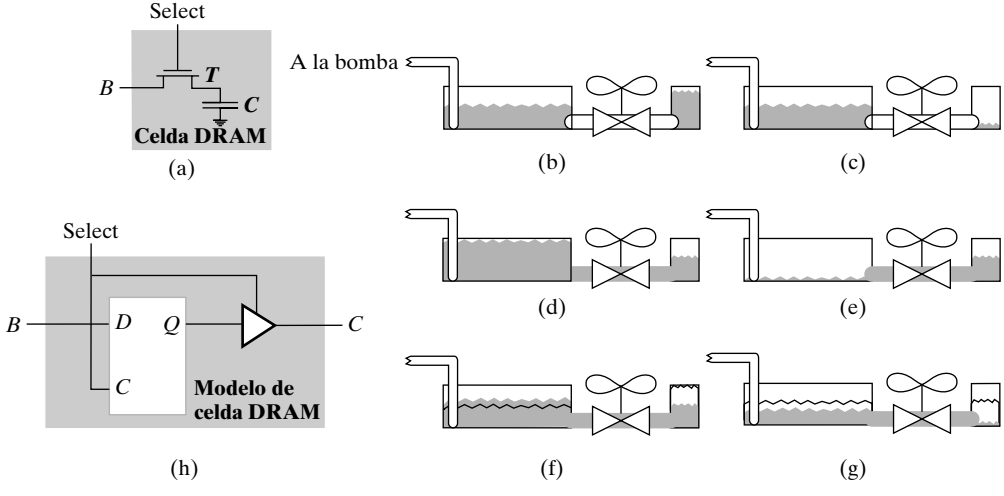
Las memorias RAM dinámicas (DRAM) dominan las aplicaciones con memorias de alta capacidad, incluyendo la memoria principal de las computadoras, debido a que proporcionan alta capacidad a bajo coste. Lógicamente, en muchos aspectos la memoria DRAM es similar a la memoria SRAM. Sin embargo, debido a los circuitos electrónicos utilizados para fabricar las celdas de almacenamiento, su diseño electrónico es considerablemente más desafiante. Además, como el nombre «dinámico» implica, el almacenamiento de la información es inherentemente temporal. Como consecuencia, la información debe «refrescarse» periódicamente para imitar el comportamiento del almacenamiento estático. Esta necesidad de refresco es la principal diferencia lógica en el comportamiento de una memoria DRAM en comparación con la SRAM. Exploraremos esta diferencia lógica examinando la celda de una memoria RAM dinámica, la lógica requerida para realizar la operación de refresco y el impacto de esta operación en el funcionamiento de la memoria del sistema.

Celda DRAM

En la Figura 9-12(a) se muestra la celda de una memoria RAM dinámica. Consiste en un condensador C y un transistor T. El condensador se usa para almacenar carga eléctrica. Si hay suficiente carga almacenada en el condensador, se puede considerar que hay almacenado un 1 lógico. Si la carga almacenada en el condensador es insuficiente, se puede considerar que hay almacenado un 0 lógico. El transistor actúa de forma muy parecida a un conmutador, de la misma forma que la puerta de transmisión presentada en el Capítulo 2. Cuando el conmutador está «abierto», la carga del condensador permanece fija aproximadamente, en otras palabras, está almacenada. Pero cuando el conmutador está «cerrado», la carga puede fluir a dentro y a fuera del condensador a través de la línea externa (B). Este flujo de carga permite escribir en la celda un 1 o un 0 y ser leída.

Para comprender las operaciones de lectura y escritura de la celda utilizaremos una analogía hidráulica con agua en lugar de carga, con un pequeño depósito en lugar del condensador y una válvula en lugar de un transistor. La línea B tiene una gran capacidad y se representa por un depósito grande y una bomba que puede llenar y vaciar el depósito rápidamente. En las Figuras 9-12(b) y 9-12(c) se representa está analogía con la válvula cerrada. Nótese que en un caso el depósito pequeño está lleno, representado un 1 y en el otro caso está vacío, representando un 0. Suponga que se va a escribir un 1 en la celda. La válvula se abre y la bomba llena el depósito grande. El agua fluye a través de la válvula llenando el depósito pequeño, como se muestra en la Figura 9-12(d). Luego se cierra la válvula dejando el depósito pequeño lleno, lo cual representa un 1. Se puede escribir un 0 utilizando el mismo tipo de procedimiento con la excepción de que la bomba vacía el depósito grande, como se muestra en la Figura 9-12(e).

Suponga ahora que queremos leer un valor almacenado y ese valor es 1, que se corresponde con un depósito lleno. Con el depósito grande a un nivel conocido intermedio, se abre la válvula. Puesto que el depósito pequeño está lleno, el agua fluye del depósito pequeño al grande, incrementando su nivel de agua ligeramente, como se muestra en la Figura 9-12(f). Este incremento en el nivel se observa como la lectura de un 1 del almacenamiento del depósito. Corres-



□ **FIGURA 9-12**
Celda de memoria RAM dinámica, analogía hidráulica del funcionamiento de la celda y modelo de la celda

pondientemente, si el depósito de almacenamiento está inicialmente vacío, habrá un leve decremento en el nivel del depósito grande de la Figura 9-12(g), el cual se observa como la lectura de un 0 del depósito de almacenamiento.

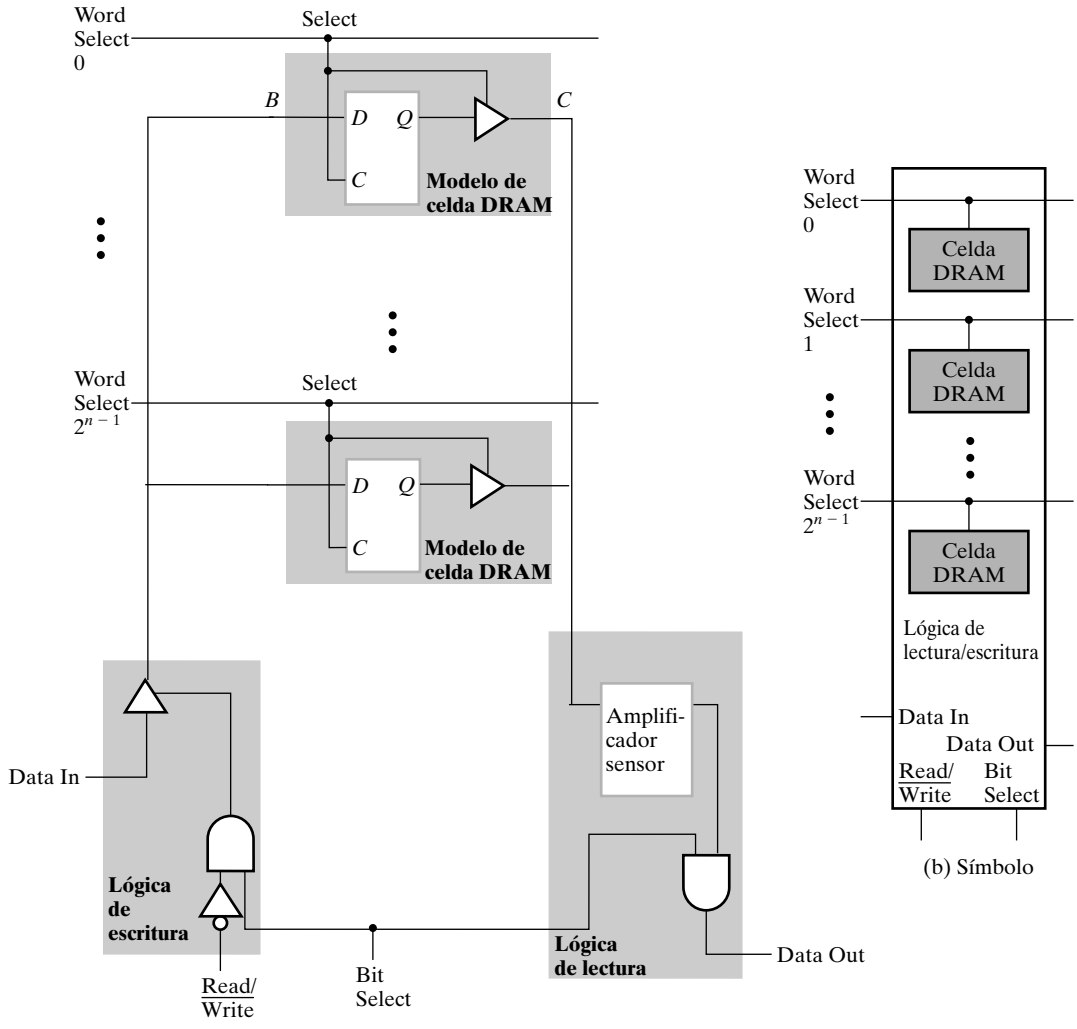
En la operación de lectura descrita, las Figuras 9-12(f) y 9-12(g) muestran que, independientemente del valor inicial almacenado en el depósito de almacenamiento, contiene ahora un valor intermedio que no provoca cambio suficiente en el nivel del depósito externo para permitir observar un 0 o un 1. Así, la operación de lectura ha destruido el valor almacenado; a esto le llamaremos *lectura destructiva*. Para permitir una lectura del valor original almacenado en un futuro, debemos *restaurar* dicho valor (es decir, devolver el depósito de almacenamiento a su nivel original). Para llevar a cabo el restablecimiento del 1 observado, el depósito grande se llena con la bomba y el depósito pequeño se llena a través de la válvula abierta. Para llevar al cabo el restablecimiento de un 0 almacenado que se ha observado, el depósito grande se vacía con la bomba y el depósito pequeño se desagua a través de la válvula abierta.

En las celdas de almacenamiento actuales hay otros caminos para el flujo de cargas. Estos caminos son análogos a pequeñas fugas en un depósito de almacenamiento. Debido a estas fugas, un depósito pequeño podrá desaguarse ocasionalmente hasta un punto en el que el incremento de nivel del depósito grande en una lectura puede no ser visto como un incremento. De hecho, si el depósito pequeño está a menos de la mitad cuando se lee, es posible que se pueda observar un decremento en el nivel del depósito grande. Para compensar estas fugas, el depósito pequeño, que almacena un 1, debe llenarse periódicamente. A esto se llama refresco del contenido de la celda. Cada celda de almacenamiento debe refrescarse antes de que su nivel haya caído a un punto tal que el valor almacenado no se pueda observar correctamente.

El funcionamiento de la memoria DRAM se ha explicado mediante una analogía hidráulica. Igual que hicimos con la memoria SRAM, empleamos un modelo lógico para la celda. El modelo se muestra en la Figura 9-12(h) es un latch tipo *D*. La entrada *C* al latch tipo *D* es Select y la entrada *D* del latch es *B*. Para modelar la salida de la celda de la memoria DRAM usamos un buffer triestado con la señal Select como entrada de control y *C* como su salida. En el circuito electrónico original de la celda de memoria DRAM de la Figura 9-12(a), *B* y *C* son la misma señal pero en el modelo lógico están separadas. Esto es necesario hacerlo en el proceso de modelado para evitar conectar las salidas de las puertas.

Tira de un bit de memoria DRAM

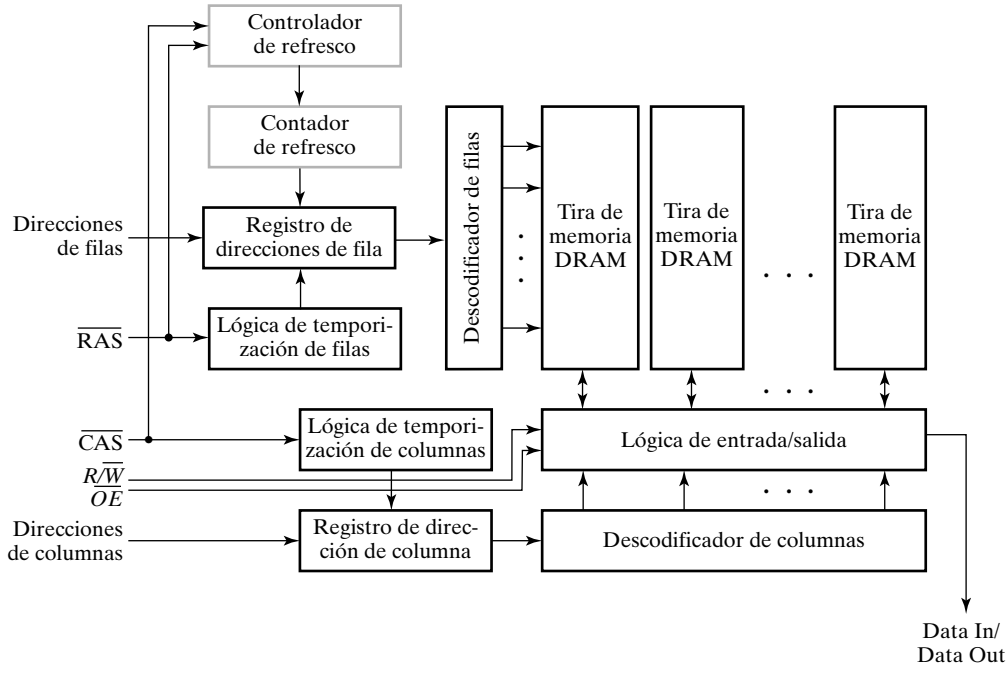
Usando el modelo lógico de la celda de memoria DRAM, vamos a construir el modelo de una tira de un bit de una memoria DRAM, como se muestra en la Figura 9-13. Este modelo es similar al de la tira de un bit de la memoria DRAM de la Figura 9-5. Es evidente que, aparte de la estructura de la celda, las dos tiras de un bit de memoria RAM son similares desde el punto de vista lógico. Sin embargo, desde el punto de vista del coste por bit, hay bastantes diferencias. La celda de una memoria DRAM está formada por un condensador más un transistor. La celda de memoria SRAM está formada típicamente por seis transistores, dando a la celda una complejidad tres veces superior, aproximadamente, a la de la memoria DRAM. Por tanto, el número de celdas de memoria SRAM, en un chip de un tamaño dado, es menor que un tercio del número de celdas en una memoria DRAM. El coste por bit de una memoria DRAM es menor que un tercio del coste por bit de una memoria SRAM, lo cual justifica el uso de memorias DRAM en memorias de gran capacidad.



(a) Diagrama lógico

□ **FIGURA 9-13**
Modelo de la tira de un bit de una memoria DRAM

Queda por discutir el refresco del contenido de la memoria DRAM. Antes de eso, necesitamos desarrollar la estructura típica usada para manejar el direccionamiento de las memorias DRAM. Puesto que muchos chips de memoria DRAM se usan para formar una memoria DRAM, queremos reducir el tamaño físico de los chips de la DRAM. Las memorias DRAM de gran capacidad necesitan 20 o más bits de dirección, lo cual supone 20 pines de dirección en cada chip. Para reducir el número de pines, las direcciones de la memoria DRAM se aplican vía serie en dos partes, la primera para la dirección de las filas y la segunda para la dirección de las columnas. Esto se puede hacer puesto que la dirección de las filas, que realiza la selección de fila, en realidad se necesita un tiempo antes que la dirección de las columnas, que es la que saca los datos para la lectura de la fila seleccionada. Para mantener las direcciones de la fila durante el ciclo de lectura o escritura, se almacena en un registro, según se muestra en la Figura 9-14. Las direcciones de las columnas también se almacenan en un registro. La señal de carga para el

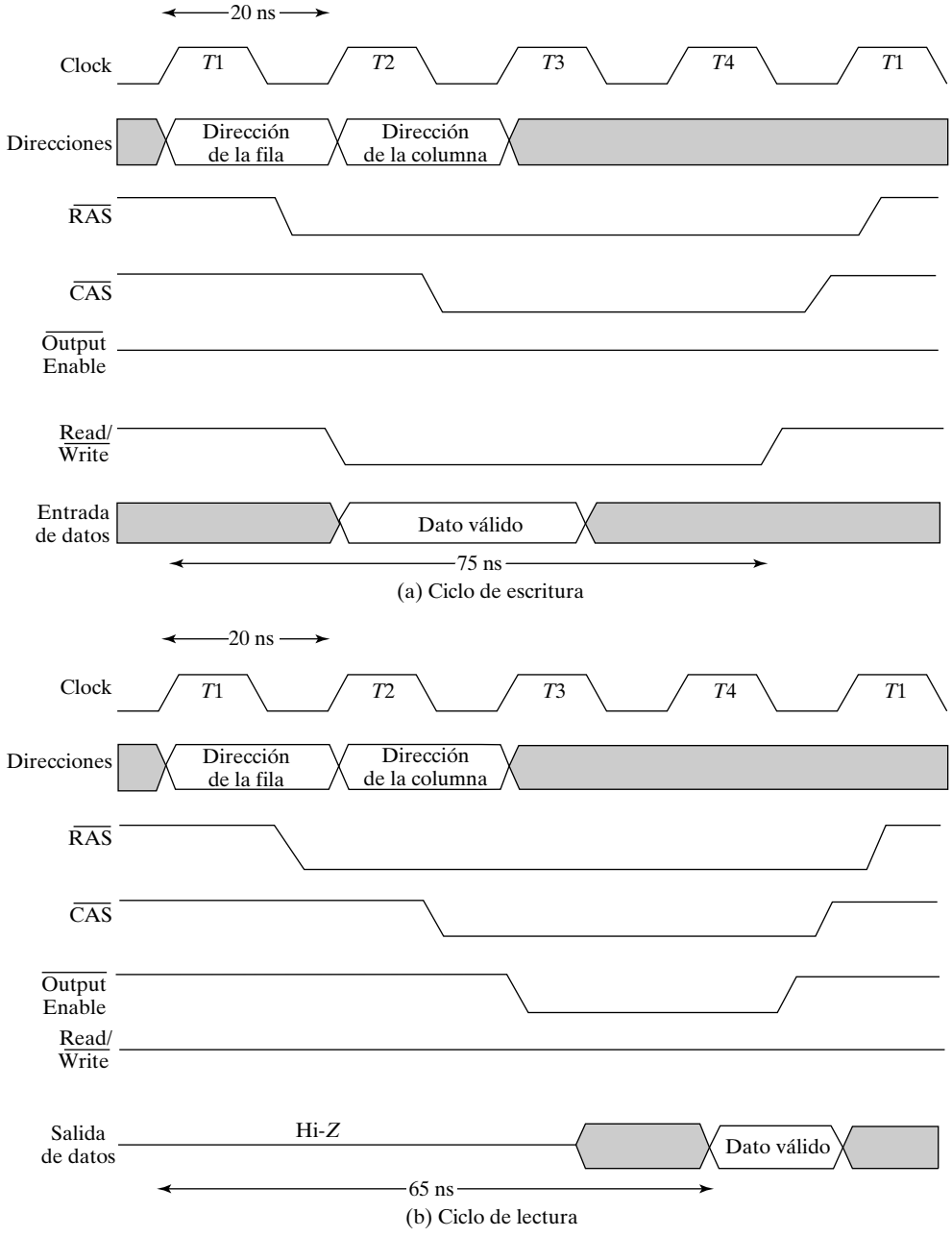


□ **FIGURA 9-14**
Diagrama de bloques de una memoria DRAM incluida su lógica de refresco

registro de direcciones de las filas es $\overline{RAS}$ (del inglés *Row Adress Strobe*) y para el registro de direcciones de las columnas es $\overline{CAS}$ (del inglés *Column Adress Strobe*). Además de estas señales de control, los chips de memoria DRAM también tienen las señales de control $R/\overline{W}$ (Read/$\overline{\text{Write}}$) y $\overline{OE}$ (del inglés *Output Enable*). Véase que en este diseño usa señales activas a nivel bajo.

En la Figura 9-15(a) aparece la temporización de las señales para las operaciones de escritura y lectura. La dirección de la fila se aplica a las entradas de dirección, y la señal RAS cambia de 1 a 0, cargando la dirección de la fila en el registro de direcciones de las filas. Esta dirección se aplica al decodificador de las direcciones de las filas y selecciona una fila de celdas de la memoria DRAM. Mientras tanto, se aplican la dirección de la columna y, luego, la señal CAS cambia de 1 a 0, cargando la dirección de la columna en el registro de direcciones de las columnas. Esta dirección se aplica al decodificador de direcciones de las columnas, seleccionando un conjunto de columnas del array de la memoria RAM, de igual tamaño que el número de bit de los datos. La entrada de un dato, con Read/$\overline{\text{Write}}$ = 0, se aplica en un intervalo de tiempo similar al de la dirección de la columna. Los bits de datos se llevan a un conjunto de líneas de bits seleccionadas por el decodificador de direcciones de columnas, para llevar estos valores a las celdas de la memoria de la columna seleccionada, escribiendo los datos nuevos en las celdas. Cuando CAS y RAS pasan a valer 1, el ciclo de escritura se ha completado y las celdas de la memoria almacenan los nuevos datos escritos. Véase que el dato almacenado en el resto de las celdas de la fila direccionada se ha restaurado.

La temporización de las señales en el ciclo de lectura es similar, como se muestra en la Figura 9-15(b). La temporización de las direcciones es el mismo. Sin embargo, no se aplican datos y Read/$\overline{\text{Write}}$ es 1 en lugar de 0. Los valores del dato en las celdas de la memoria DRAM de la fila seleccionada se aplican a las líneas y se detectan por amplificadores sensores. El decodificador de direcciones de las columnas selecciona los valores que van a ser enviados a la



□ **FIGURA 9-15**
Temporización para las operaciones de lectura y escritura de una memoria DRAM

salida de datos, habilitada por $\overline{OE}$. Durante la operación de lectura, todos los valores de la fila seleccionada se restauran.

Para proporcionar el refresco está la lógica adicional en el diagrama de bloques de la Figura 9-14 (rectángulos coloreados). Hay un contador de refresco y un controlador de refresco. El contador de refresco se usa para proporcionar la dirección de la fila de las celdas de la me-

moria DRAM a refrescar. Esto es fundamental en los modos de refresco que necesita la dirección a ser proporcionada desde el interior del chip de la memoria DRAM. El contador de refresco avanza en cada ciclo de refresco. Según el número de bits del contador, cuando alcanza el valor $2^n - 1$, donde n es el número de filas del array de la memoria, pasa a 0 en el siguiente ciclo de refresco. Las formas estándar en la que el refresco se efectúa y el tipo de refresco correspondiente son los siguientes:

1. **RAS only refresh**. La dirección de una columna se coloca en las líneas de dirección y la señal RAS se pone a 0. En este caso, la dirección a refrescar debe proceder del exterior del chip de memoria DRAM, típicamente de un circuito integrado llamado controlador de DRAM.
2. **CAS before RAS refresh**. La señal CAS cambia de 1 a 0 y seguida de un cambio de 1 a 0 en la señal RAS. Se pueden llevar a cabo ciclos de refresco adicionales cambiando la señal RAS sin cambiar la señal CAS. En este caso, las direcciones de refresco proceden del contador de refresco, que se incrementa después del refresco para cada ciclo.
3. **Hidden refresh**. Siguiendo una escritura o lectura normal, la señal CAS se deja a 0 y la señal RAS se activa cíclicamente, efectuando un refresco del tipo *CAS before RAS refresh*. Durante un *hidden refresh*, la salida del dato de la anterior lectura permanece válido. De esta forma, el refresco permanece oculto. Desafortunadamente, el tiempo empleado por el *hidden refresh* es significativo, de forma que se retrasa la siguiente operación de lectura y escritura.

En todos los casos, tenga en cuenta que el inicio del refresco se controla externamente usando las señales $\overline{RAS}$ y $\overline{CAS}$. Cada fila de un chip de memoria DRAM necesita un refresco dentro de un tiempo de refresco máximo especificado, típicamente en el rango de 16 a 64 milisegundos (ms). Los refrescos pueden realizarse en puntos espaciados uniformemente, llamándose entonces refresco en modo distribuido. Alternativamente, todos los refrescos se realizan uno después de otro, llamándose refresco en modo ráfaga. Por ejemplo, una memoria DRAM de 4 M × 4 tiene un tiempo de refresco de 64 ms y tiene 4096 filas para refrescar. La cantidad de tiempo para realizar un único refresco es de 60 ns y el intervalo de refresco en modo distribuido es 64 ms/4096 = 15,6 microsegundos (μs). El tiempo total de refresco de 0.25 ms se saca del intervalo de 64 ms del intervalo de refresco. Durante los ciclos de refresco no se pueden efectuar operaciones de lectura ni de escritura en la memoria DRAM. Puesto que la ráfaga de refresco podría parar la operación de la computadora durante un periodo bastante largo, el modo de refresco más utilizado es el distribuido.

9-6 TIPOS DE MEMORIA DRAM

En las dos últimas décadas la capacidad y la velocidad de las memorias DRAM se han incrementado significativamente. La demanda de mayor velocidad ha dado lugar a la evolución de varios tipos de memoria DRAM. En la Tabla 9-2 se enumeran algunos tipos de memoria DRAM junto a una breve descripción. De los tipos de memoria enumerados, los dos primeros tipos han sido reemplazados en el mercado por memorias SDRAM y RDRAM más avanzadas. Al existir una presentación sobre códigos de corrección de errores (ECC) para memorias en la página web del libro, nos centraremos en las memorias DRAM síncronas, memorias DRAM de doble velocidad (*double data rate synchronous* DRAM) y memorias DRAM Rambus©. Antes de considerar estos tres tipos de memoria DRAM se presentaran brevemente algunos conceptos básicos.

□ **TABLA 9-2**
Tipos de DRAM

Tipo	Abreviatura	Descripción
Fast Page Mode DRAM	FPM DRAM	Toma la ventaja del hecho de que, cuando se accede a una fila, todos sus valores están disponibles para leerse. Cambiando la dirección de la columna se pueden leer los datos de diferentes direcciones sin necesidad de volver a poner la dirección de la fila y sin tener que esperar por el retardo asociado con la lectura de la fila de celdas si la porción de la fila coincide con las direcciones
Extended Data Output DRAM	EDO DRAM	Aumenta el tiempo que la DRAM mantiene los valores de los datos en su salida, permitiendo que la CPU realice otras tareas durante el acceso puesto que sabe que el dato estará todavía disponible.
Synchronous DRAM	SDRAM	Funciona con un reloj en lugar de operar asíncronamente. Esto permite una más estrecha interacción entre la memoria y la CPU ya que la CPU conoce exactamente cuando los datos estarán disponibles. La memoria SDRAM usa la ventaja de la disponibilidad del valor de la fila y divide la memoria en distintos bancos, permitiendo el acceso solapado.
Double Data Rate Synchronous DRAM	DDR SDRAM	Igual que la memoria SDRAM excepto que la salida de los datos se proporciona tanto en el flanco de subida como en el de bajada.
Rambus© DRAM	RDRAM	Una tecnología propietaria que proporciona una muy alta velocidad de acceso usando un bus relativamente estrecho.
Error-Correcting Code	ECC	Se puede aplicar a la mayoría de los anteriores tipos de memorias DRAM para corregir errores en los datos de un bit y detectar los errores de dos bits.

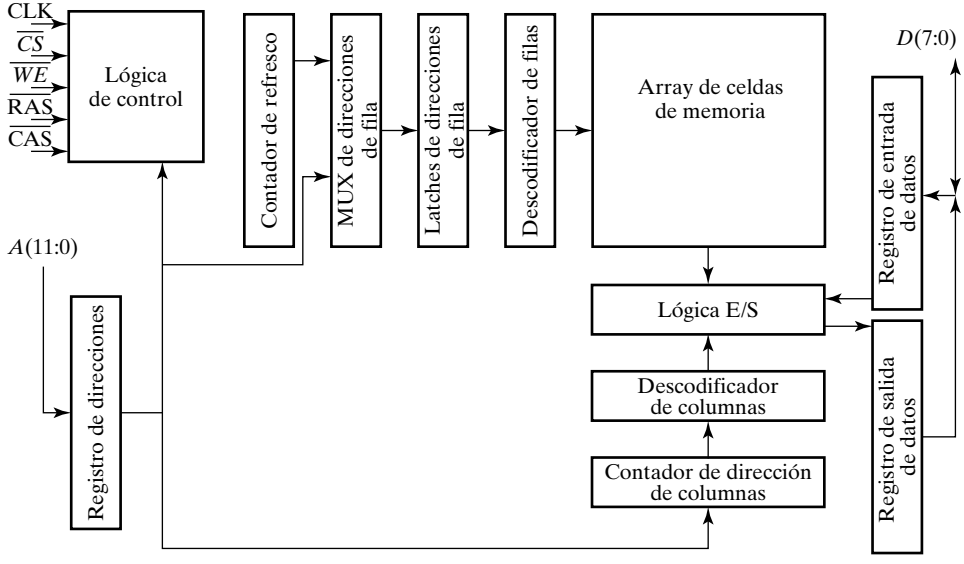
Primero, hemos de indicar que estos tres tipos de memoria DRAM funcionan bien debido al particular entorno en el que trabajan. En los modernos sistemas de alta velocidad, el procesador interactúa con la memoria DRAM dentro de una memoria jerárquica. La mayoría de las instrucciones y datos para el procesador se extraen de los dos niveles más bajos de la jerarquía, las cachés de primer y segundo nivel (L1 y L2 respectivamente). Estas memorias son, en comparación, más pequeña que las estructuras de memoria basadas en memorias SRAM, que se tratan con detalle en el capítulo 14. En nuestro estudio, la cuestión clave es que la mayoría de las lecturas de la DRAM no se realizan directamente por la CPU, en su lugar, se realizan lecturas

iniciales que llevan los datos y las instrucciones a estas cachés. Las lecturas se efectúan sobre un bloque de direcciones consecutivas cuya información se lleva a las cachés. Por ejemplo, realizar una lectura de 16 bytes de las direcciones 000000 a la 00000F. A esto se le llama ráfaga de lectura. En ráfagas de lectura el parámetro importante es la velocidad efectiva de lectura de bytes, que depende de las lecturas de las direcciones consecutivas, más que el tiempo de acceso. Según este parámetro, los tres tipos de memorias presentadas proporcionan un funcionamiento muy rápido.

Segundo, la efectividad de estos tres tipos de memoria DRAM depende del principio fundamental involucrado en el funcionamiento de las memorias DRAM, la lectura de todos y cada uno de los bits de una fila en cada operación de lectura. Este principio implica que todos los bits de la fila están disponibles después de una lectura usando esa fila si estos deben accederse. Con estos dos conceptos en mente, se puede presentar la memoria DRAM síncrona (SDRAM).

Memoria síncrona DRAM (SDRAM)

El uso de transferencias sincronizadas con el reloj diferencia a la memoria SDRAM de la DRAM convencional. En la Figura 9-16 aparece el diagrama de bloques de un circuito de memoria SDRAM de 16 megabytes. Las entradas y las salidas difieren en algo de las del diagrama de bloques de la memoria DRAM de la Figura 9-14 con la excepción de la presencia del reloj para sincronizar las operaciones. Internamente, hay ciertas diferencias. Puesto que la memoria SDRAM tiene una apariencia externa síncrona, tiene registros en las entradas de dirección y en las entradas y salidas de datos. Además se le ha incorporado un contador de direcciones de columna, que es la clave del funcionamiento de la memoria SDRAM. La lógica de control puede parecer similar pero, en este caso, es mucho más complicada ya que tiene una palabra de control de modo que se puede cargar del bus de direcciones. Suponiendo una memoria de 16 MB, el array de celdas de la memoria contiene 134 217 728 bits y es casi cuadrada, con

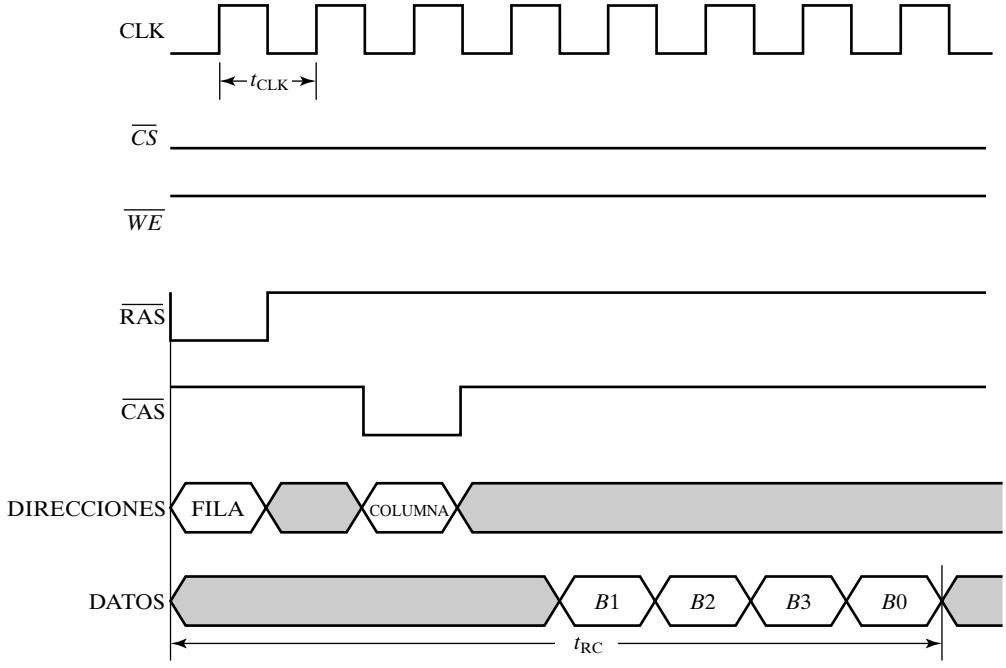


□ **FIGURA 9-16**
Diagrama de bloques de un memoria SDRAM de 16 MB

8 192 filas y 16 384 columnas. Tiene 13 bits de direcciones para las filas. Como tiene 8 bits por byte, el número de columnas direccionables es de 16 384 dividido por 8, es decir, 2048, es decir, hacen falta 11 bits para direccionar las columnas. Véase que 13 más 11 es igual a 24, que es el número correcto de bits de direcciones para una memoria de 16 MB.

Como en la DRAM normal, la memoria SDRAM aplica primero las direcciones de las filas seguidas de las direcciones de las columnas. Sin embargo, la temporización es algo diferente y se emplean algunos nuevos términos. Antes de realizar una operación de lectura de una determinada fila, todos los 2048 bytes de la fila especificada por la dirección se leen internamente y se almacenan en la lógica de entrada/salida. Este paso necesita internamente algunos ciclos de reloj. A continuación, la lectura se realiza con la dirección de la columna aplicada. Después de un retardo adicional de algunos ciclos de reloj, los bytes de datos empiezan a aparecer en la salida, uno por cada ciclo de reloj. El número de bytes que aparecen, la longitud de la ráfaga, ha sido cambiado cargando la palabra de control en la lógica de control desde la entrada de direcciones.

En la Figura 9.17 se muestra la temporización de un ciclo de lectura de una ráfaga de longitud igual a 4. La lectura comienza con la aplicación de la dirección de la fila y la habilitación de la dirección de la fila (*row address strobe*, RAS), que hace que la dirección de la fila sea capturada en el registro de dirección y que comience la lectura de la fila. Durante los siguientes dos ciclos de reloj tiene lugar la lectura de la fila. Durante el tercer ciclo de reloj se pone la dirección de las columnas y su habilitación (*column address strobe*, CAS), con la dirección de la columna capturada en el registro de direcciones y la lectura del primer byte iniciada. El byte ya está disponible para leerse de la memoria SDRAM en el flanco positivo dos ciclos después. El segundo, tercer y cuarto byte están disponibles para su lectura en los siguientes flancos de reloj. En la Figura 9-17 se puede observar que los bytes se presentan en el orden 1, 2, 3 y 0. Esto es así porque, en la identificación de la dirección de la columna, el byte que necesita inmediata-



□ **FIGURA 9-17**
Diagrama de tiempos de una memoria SDRAM

mente la CPU, los dos últimos bits son 01. Los siguientes bytes que aparecen dependen de la cuenta ascendente del módulo la longitud de la ráfaga que realiza el contador de direcciones de las columnas, dando direcciones que terminan en 01, 10, 11 y 00, permaneciendo el resto de bits fijos.

Es interesante comparar la velocidad de lectura de los bytes de una memoria SDRAM y de una memoria básica DRAM. Suponiendo que el tiempo del ciclo de lectura, t_{RC}, para una memoria DRAM básica es de 60 ns y que el periodo de reloj, t_{CLK}, es de 7.5 ns. La velocidad de lectura de una memoria DRAM básica es de un byte cada 60 ns o 16,67 MB/s. Para la SDRAM de la Figura 9-17, se necesitan 8.0 ciclos de reloj, es decir, 60 ns, para leer 4 bytes, dando una velocidad de 66.67 MB/s. Si la ráfaga es de ocho en lugar de cuatro, se necesita un tiempo de ciclo de lectura de 90 ns, dando una velocidad de 88.89 MB/s. Finalmente, si la ráfaga comprende los 2048 bytes de la fila de la memoria SDRAM, el tiempo del ciclo de lectura es $60 + (2048 - 4) \times 7.5 = 15.390$ ns, dando una tasa de transferencia de 133.07 MB/s, cercano al límite de un byte por ciclo de reloj de 7.5 ns.

Memoria SDRAM de doble tasa de transferencia de datos (DDR SDRAM)

El segundo tipo de memoria DRAM es la SDRAM de doble tasa de transferencia de datos (*double data rate* SDRAM, DDR SDRAM) supera el límite anteriormente presentado sin tener que decrementar el periodo de reloj. En cambio puede proporcionar dos bytes de datos por cada ciclo de reloj usando tanto el flanco de subida como el flanco de bajada. En la Figura 9-17 se leen cuatro bytes, uno por cada ciclo de reloj. Usando ambos flancos de reloj, se pueden transferir ocho bytes en el mismo tiempo de ciclo de lectura, t_{RC}. Para un periodo de reloj de 7.5 ns, la tasa de transferencia dobla la del ejemplo dando lugar a 266.24 MB/s.

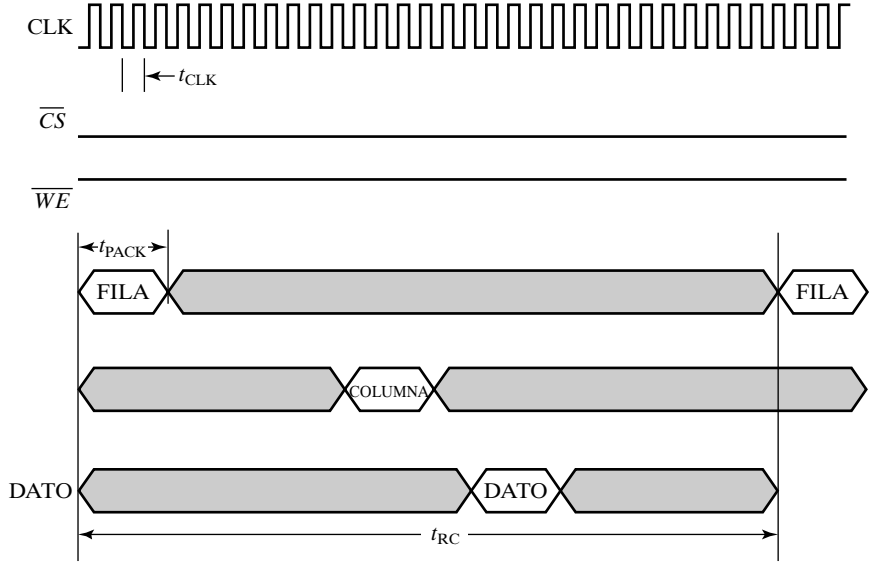
Se pueden aplicar otras técnicas básicas para incrementar la tasa de transferencia de bytes. Por ejemplo, en lugar de tener un dato de un solo byte se puede tener un chip de memoria SDRAM con una longitud del dato de entrada/salida de cuatro bytes (32 bits). Esto proporciona una tasa límite de transferencia de 1066 GB/s con un reloj con un periodo de 7.5 ns. Con 8 bytes se alcanzaría una tasa de 2130 GB/s.

La tasa de transferencia conseguida en los ejemplos son los límites superiores. Si un acceso se ha de realizar usando diferentes filas de la RAM, el retardo desde la aplicación de pulso RAS a la lectura del primer byte de dato es significativo y lleva a rendimientos bastante por debajo del límite. Esto se puede compensar parcialmente dividiendo la memoria en varios bancos donde cada uno realiza la lectura de la columna independientemente. Con tal de que la dirección de la fila y del banco estén disponibles con bastante antelación, la lectura de la fila se puede realizar sobre uno o más bancos mientras el dato de la columna activa se está aún transfiriendo. Cuando la lectura de la columna de la fila activa se completa, el dato puede estar potencialmente disponible inmediatamente de otros bancos, permitiendo un flujo ininterrumpido de datos de la memoria. Esto permite conseguir unas tasas de transferencia cercanas al límite. Sin embargo, debido al hecho de que puede ocurrir que se accedan secuencialmente a varias filas del mismo, esta tasa máxima nunca se alcanza.

Memoria RAMBUS© DRAM (RDRAM)

El último tipo de memoria DRAM que se va a presentar es la memoria RAMBUS DRAM. Los circuitos integrados RDRAM se han diseñado para ser integrados en la memoria de un sistema

que usa un bus basado en paquetes para la interacción de los circuitos de memoria RDRAM y el bus de memoria con el procesador. Los principales componentes del bus son: una ruta de 3 bits para las direcciones de las filas, una ruta de 5 bits para las direcciones de las columnas y una ruta de 16 o 18 bits para los datos. El bus es síncrono y efectúa transferencias en ambos flancos de reloj, como en la memoria DDR SDRAM. La información en las tres rutas anteriormente presentadas se transfiere en paquetes durante cuatro ciclos de reloj, es decir, se realizan 8 transferencias de paquetes. El numero de bits por paquete de cada una de las rutas es de 24 bits para el paquete de direcciones de las filas, 40 bits para los paquetes de direcciones de las columnas y 128 o 144 bits para los paquetes de datos. El paquete más grande incluye 16 bits de paridad para realizar códigos de corrección de errores. Los circuitos RDRAM emplean el concepto de varios bancos de memoria, como se mencionó anteriormente, para proporcionar la posibilidad de realizar accesos concurrentes de diferentes direcciones de filas. La memoria RDRAM utiliza la técnica habitual de fila activada en la que se lee la fila direccionada de la memoria. A partir de esta fila de datos, la dirección de la columna se usa para seleccionar las parejas de bytes en el orden en que deben ser transmitidos en el paquete. En la Figura 9-18 se muestra una temporización típica de un acceso de lectura de una memoria RDARM. Como consecuencia del sofisticado diseño electrónico del sistema de memoria RAMBUS, consideramos un periodo de reloj de 1.875 ns. Así, el tiempo de transmisión de un paquete es $t_{\text{PACK}} = 4 \times 1.875 = 7.5$ ns. El tiempo del ciclo de acceso para un solo paquete de datos de 8 parejas de bytes o 16 bytes es de 266.67 MB/s. Si se accede a cuatro paquetes de un byte de la misma fila, la tasa se incremente a 1.067 GB/s. Para leer todo el contenido de una fila de una memoria RDRAM de 2048 bytes, el tiempo de acceso se incrementa en $60 + (2048 - 64) \times 1.875/4 = 990$ ns o a la tasa límite de transferencia de bytes de $2048/(990 \times 10^{-9}) = 2.069$ MB/s, aproximándose al límite ideal de de 4/1.875 ns o 2.133 GB/s.



□ **FIGURA 9-18**
Diagrama de tiempos de una RDRAM de 16 MB

9-7 ARRAYS DE CIRCUITOS INTEGRADOS DE MEMORIAS DINÁMICAS RAM

Muchos de los principios de diseños utilizados en la Sección 9-4 para hacer *arrays* de memorias SRAM se aplican al diseño de *arrays* DRAM. Sin embargo existen diversos requerimientos diferentes para controlar y direccionar los *arrays* de las memorias DRAM. Estos requerimientos se llevan a cabo mediante un controlador de memorias DRAM. Las funciones realizadas por dicho controlador son las siguientes:

1. realiza la separación de las direcciones en direcciones de las filas y las direcciones de las columnas, proporcionando la temporización necesaria,
2. proporciona las señales $\overline{RAS}$ y $\overline{CAS}$ con su temporización correcta para las operaciones de lectura, escritura y refresco.
3. lleva a cabo las operaciones de refresco en los intervalos apropiados, y
4. proporciona las señales de status al resto del sistema (por ejemplo, indica cuándo la memoria está ocupada realizando su intervalo de refresco).

El controlador de la memoria DRAM es un circuito secuencial complejo sincronizado con el reloj exterior de la CPU para su funcionamiento.

9-8 RESUMEN DEL CAPÍTULO

Las memorias son de dos tipos: memorias de acceso aleatorio (RAM) y memorias de sólo lectura (ROM). En ambos tipos aplicamos una dirección para leer o escribir un dato. Las operaciones de lectura y escritura han de cumplir ciertos pasos con sus parámetros temporales, incluyendo el tiempo de acceso y el tiempo del ciclo de escritura. Las memorias pueden ser estáticas o dinámicas y volátiles o no volátiles. Internamente, un circuito de una memoria RAM está formado por un array de celdas RAM, decodificadores, circuitos de escritura, circuitos de lectura y circuitos de salida. La combinación de estos circuitos de escritura, lectura y las celdas RAM asociadas se pueden modelar lógicamente como una tira de memoria RAM de un bit. Las tiras de memoria RAM de un bit se pueden combinar para formar *arrays* de celdas de dos dimensiones que, junto con los decodificadores y los circuitos de salida, forman la base para un chip de memoria RAM. Los circuitos de salida utilizan *buffers* tri-estado para facilitar la conexión de un array de chips de memoria con una lógica adicional reducida. En las memorias DRAM, debido a la necesidad de refresco, se debe utilizar circuitería adicional para llevarlo a cabo así como para utilizar *arrays* de chips. Como consecuencia de la necesidad de tener memorias con accesos más rápidos se han desarrollado nuevos tipos de memorias DRAM. Los tipos más recientes de memorias de alta velocidad DRAM emplean interfaces síncronas que usan un reloj para el control de los accesos de la memoria.

Los códigos de detección y corrección de errores, basados frecuentemente en Códigos Hamming, se usan para detectar o corregir errores de los datos almacenados en las memorias RAM. Existe material de la primera edición que cubre este tipo de códigos disponible en la dirección de Internet: http://www.librosite.net/Mano.

El materia que cubre las memorias en VHDL y Verilog está disponible en la dirección de Internet: http://www.librosite.net/Mano.

REFERENCIAS

1. WESTE, N. H. E., and ESHRAGHIAN, K.: *Principles of CMOS VLSI Design: A Systems Perspective,* 2nd ed. Reading, MA: Addison-Wesley, 1993.
2. Micron Technology, Inc. *Micron 256 Mb: x4, x8, x16 SDRAM*. www.micron.com, 2002.
3. Micron Technology, Inc. *Micron 64 Mb: x32 DDR SDRAM*. www.micron.com, 2001.
4. SOBELMAN, M.: «Rambus Technology Basics», *Rambus Developer Forum*. Rambus, Inc., October 2001.
5. Rambus, Inc. *Rambus Direct RDRAM 128/144-Mbit (256x16/18x32s) - Preliminary Information,* Documento DL0059 Versión 1.11.

PROBLEMAS

El signo (+) indica problemas más avanzados y el asterisco (*) indica que hay una solución disponible en la dirección de Internet: http://www.librosite.net/Mano.

9-1. *Las siguientes memorias se especifican mediante el número de palabras y el número de bits por palabra ¿Cuántas líneas de dirección y líneas de entrada/salida de datos se necesitan en cada caso?

(a) 16 K × 8
(b) 256 K × 16
(c) 64 M × 32
(d) 2 G × 8.

9-2. Indique el número de bits almacenados en las memorias enumeradas en el Problema 9-1.

9-3. *La palabra número $(835)_{10}$ de la memoria mostrada en la Figura 9-2 contiene el dato binario equivalente a $(15\ 103)_{10}$. Indique la dirección de 10 bits y el contenido de 16 bits de la palabra.

9-4. Un chip de memoria RAM de 64 K × 16 utiliza una descodificación coincidente dividiendo el decodificador interno en selección de fila y selección de columna.

(a) Suponiendo que el array de celdas RAM es cuadrado ¿Cuál es el tamaño de cada decodificador y cuántas puertas AND se necesitan para la descodificación de una dirección?
(b) Determine que las líneas de selección de fila y de columna están habilitadas cuando la dirección de entrada es el valor binario equivalente de $(32\ 000)_{10}$.

9-5. Suponga que el decodificador más grande que se puede usar en un chip de memoria de $m \times 1$ RAM tiene 13 entradas de dirección y que se emplea descodificación coincidente. Para construir chips de memoria RAM que contengan más de *m* palabras de 1 bit, se incluyen *arrays* múltiples de celdas RAM, cada una con su decodificador y circuitos de lectura/escritura.

(a) Con la restricción dada del decodificador ¿cuántos *arrays* de celdas RAM se necesitan para construir un chip de memoria RAM de 512 M × 1?
(b) Muestre qué decodificador es necesario emplear para seleccionar entre los diferentes *arrays* de memoria RAM de la memoria y sus conexiones a los bits de direcciones y los decodificadores de las columnas.

9-6. Una memoria DRAM tiene 14 pines de dirección y su dirección de columna es de un bit. ¿Cuántas direcciones en total tiene la memoria DRAM?

9-7. Una memoria DRAM de 256 Mb, que utiliza datos de 4 bits, tiene igual longitud de direcciones de filas y de columnas. ¿Cuántos pines de dirección tiene dicha memoria?

9-8. Una memoria DRAM tiene un intervalo de refresco de 128 ms y 4096 filas. ¿Cuál es la duración del intervalo entre refresco en el modo de refresco distribuido? ¿Cuál es el número mínimo de pines de dirección de la memoria DRAM?

9-9. ***(a)** ¿Cuántos chips de memoria RAM de 128 × 16 se necesitan para conseguir una memoria de 1 M bytes de capacidad?

(b) ¿Cuántas líneas de dirección se necesitan para direccional 1 M bytes? ¿Cuántas de estás líneas están conectadas a las entradas de dirección de todos los chips?

(c) ¿Cuántas líneas se deben descodificar para generar el *chip select*? Especifique el tamaño del decodificador.

9-10. Usando el chip de memoria RAM de capacidad 64 K × 8 de la Figura 9-9 más un decodificador, construya el diagrama de bloques de una memoria RAM de 512 K × 16.

9-11. Explique cómo utiliza la memoria SDRAM la ventaja del *array* de dos dimensiones para conseguir altas tasas de transferencia de datos.

9-12. Explique cómo se consigue doblar la tasa de transferencia de una memoria DDRAM en comparación con una memoria SDRAM.

CAPÍTULO

10

FUNDAMENTOS DEL DISEÑO DE PROCESADORES

En el Capítulo 7 se presentó la división de un diseño en una ruta de datos, que realiza las microoperaciones, y una unidad de control, que determina la secuencia de las microoperaciones. En este capítulo definimos una ruta de datos de un procesador genérico que realiza las microoperaciones de transferencia de registros y sirve como marco de trabajo para el diseño detallado de la lógica de proceso. El concepto de palabra de control proporciona un nexo entre la ruta de datos y la unidad de control asociada a ella.

La ruta de datos genérica combinada con la unidad de control y la memoria forma un sistema programable, en este caso, un sencillo procesador. Se presenta el concepto de una arquitectura con un conjunto de instrucciones (*Instruction Set Architecture*, ISA) como medio de especificación de un procesador. Para realizar una arquitectura ISA se combinan una unidad de control y una ruta de datos genérica para formar una CPU (*Central Processing Unit*). Además, como es un sistema programable, también se incluyen las memorias para el almacenamiento de programas y datos. Se consideran dos tipos de procesadores con dos unidades de control diferentes. El primero tiene dos memorias, una para las instrucciones y otra para los datos, y realiza todas sus operaciones en un solo ciclo de reloj. El segundo procesador tiene una única memoria, tanto para las instrucciones como para los datos, y una arquitectura más compleja que necesita varios ciclos de reloj para realizar sus operaciones.

En la computadora genérica presentada al principio del Capítulo 1, las transferencias de registro, las microoperaciones, los buses, las rutas de datos y sus componentes, y las palabras de control se usan ampliamente. Así mismo, las unidades de control aparecen en la mayoría de los bloques digitales de un procesador genérico. El diseño de unidades de procesamiento que presentan unidades de control, que interactúan con las rutas de datos, tiene su más grande repercusión dentro del procesador genérico, en la CPU y en la FPU del circuito procesador. Estos dos componentes contienen grandes rutas de datos que realizan el procesamiento. La CPU y la FPU llevan a cabo sumas, restas y las otras operaciones especificadas por el conjunto de instrucciones.

10-1 INTRODUCCIÓN

En este capítulo se presentan los procesadores y su diseño. La especificación de un procesador consiste en una descripción de su apariencia frente al programador a nivel más bajo, su arquitectura de conjunto de instrucciones (ISA). A partir de la ISA se formula una descripción de alto nivel del hardware del procesador, llamada arquitectura del procesador. Esta arquitectura, para un procesador sencillo, se divide típicamente en una ruta de datos y una unidad de control. La ruta de datos se define mediante tres componentes básicos:

1. un conjunto de registros
2. las microoperaciones que se efectúan sobre los datos almacenados en los registros, y
3. el interfaz de control.

La unidad de control proporciona las señales que controlan las microoperaciones efectuadas en la ruta de datos y en otros componentes del sistema, como las memorias. Además, la unidad de control gobierna su propia operación, determinando la secuencia de eventos que suceden. Esta secuencia puede depender de los resultados de la ejecución de la actual microoperación y de las pasadas. En procesadores más complejos se pueden encontrar varias unidades de control y varias rutas de datos.

Para encontrar una base inicial para el diseño de procesadores, ampliaremos las ideas del Capítulo 7 para el diseño de las rutas de datos. Concretamente consideramos una ruta de datos genérica, una que se pueda usar, en algunos casos modificada, en todos los diseños de procesadores considerados en adelante en este texto. Estos futuros diseños muestran cómo una determinada ruta de datos se puede usar para realizar diferentes arquitecturas de conjunto de instrucciones, simplemente combinando la ruta de datos con diferentes unidades de control.

10-2 RUTAS DE DATOS

En lugar de tener un registro concreto que realiza directamente sus microoperaciones, los sistemas del procesador emplean cierto número de registro de almacenamiento conjuntamente con una unidad de operación compartida llamada unidad aritmético-lógica, abreviadamente ALU (del inglés *arithmetic-logic unit*). Para realizar una microoperación, el contenido de unos registros fuentes concretos se aplican a las entradas de la ALU que tienen compartida. La ALU realiza una operación y el resultado de esta operación se transfiere a un registro destino. Como la ALU es un circuito combinacional, la totalidad de la operación de transferencia de registro a partir de los registros fuentes, a través de la ALU, hasta los registros de destino se realiza en un solo ciclo de reloj. Las operaciones de desplazamiento se realizan frecuentemente en una unidad aparte, aunque algunas veces, este tipo de operaciones también se lleva a cabo en la ALU.

Resaltar que la combinación de un conjunto de registros y una ALU compartida y sus interconexiones forman la ruta de datos de un sistema. El resto de este capítulo hace referencia a la organización y diseño de rutas de datos y de las unidades de control utilizadas para realizar un procesador sencillo. Se llevará a cabo una ALU concreta para mostrar el proceso involucrado en la realización de circuitos combinacionales complejos. También diseñaremos un desplazador, combinaremos las señales de control en una palabra de control y, a continuación, añadiremos unidades de control para diseñar dos procesadores diferentes.

La ruta de datos y la unidad de control son dos partes del procesador o CPU de una computadora. Junto con los registros, la ruta de datos contiene la lógica digital necesaria que realiza diversas microoperaciones. Esta lógica digital está formada por buses, multiplexores, descodifica-

dores y demás circuitos de procesamiento. Cuando se incluyen en la ruta de datos un gran número de registros, lo más conveniente es conectarlos mediante uno o más buses. Los registros de una ruta de datos interactúan mediante transferencias de datos así como en la ejecución de varios tipos de microoperaciones. En la Figura 10-10 se muestra una ruta de datos con cuatro registros, una ALU y un desplazador. Las señales con nombre en azul, relacionadas con la Figura 10-10, se describirán en la Sección 10-5. Las señales con nombre en negro se usan aquí para describir los detalles de la Figura 10-1. Cada registro se conecta con dos multiplexores para formar los buses de entrada *A* y *B* a la ALU y al desplazador. Las entradas de selección de cada multiplexor, *A* select y *B* select, seleccionan un registro para el bus correspondiente. Para el bus *B* hay un multiplexor adicional, MUX *B*, de manera que se pueden introducir valores constantes en la ruta de datos desde el exterior utilizando la entrada «*Constant In*». El Bus *B* también está conectado a la salida «*Data Out*», para mandar datos al exterior de la ruta de datos para otros componentes del sistema, como memorias o bloques de entrada/salida.

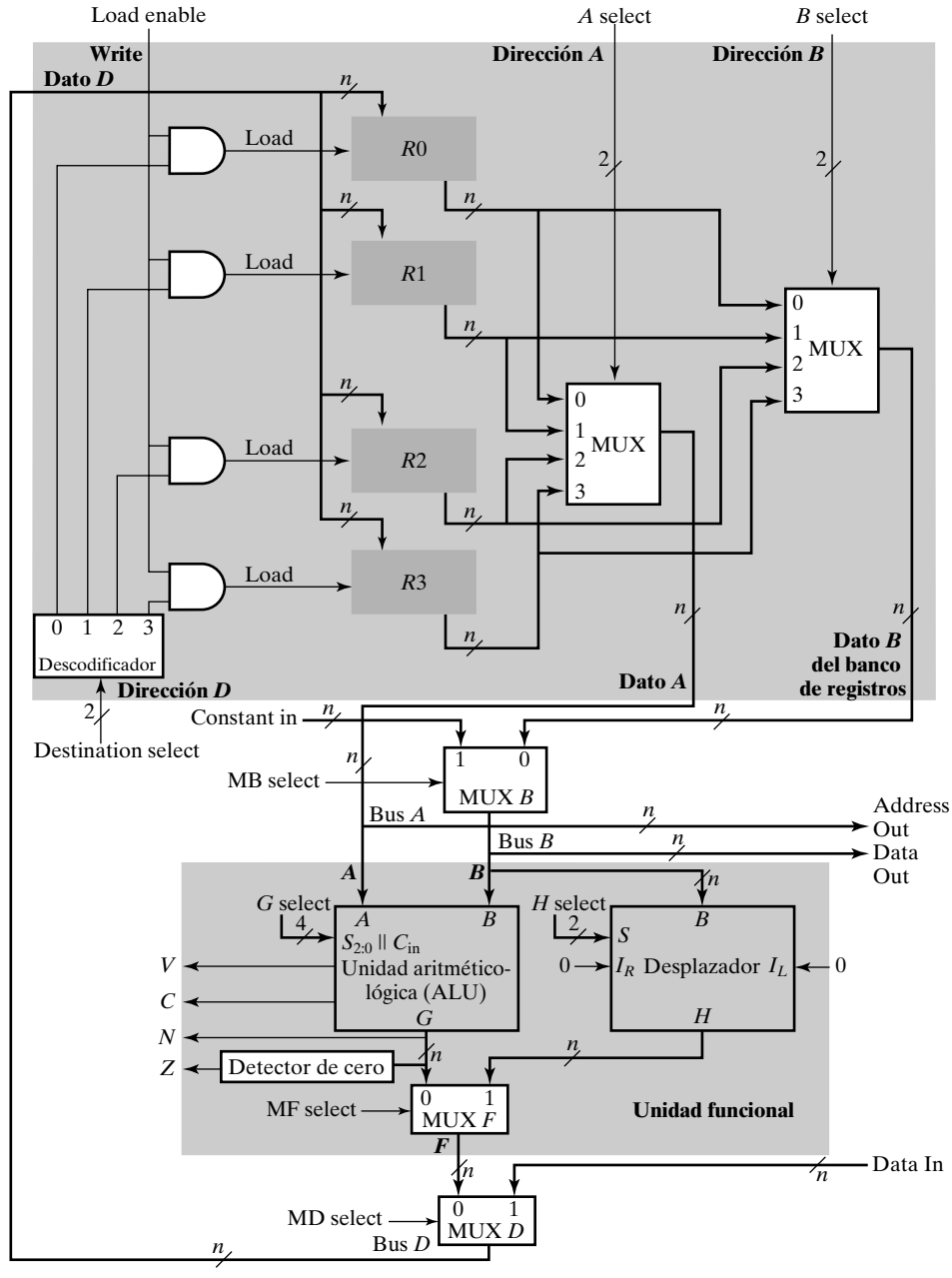
Las microoperaciones aritméticas y lógicas se efectúan sobre los operandos de los buses *A* y *B*. Las entrada de selección *G* seleccionan la microoperación a realizar por la ALU. Las microoperaciones de desplazamiento se llevan a cabo sobre los datos del Bus *B* por el desplazador. La entrada de selección *H* pasa el operando del Bus *B* directamente a través del desplazador hasta su salida o selecciona una microoperación de desplazamiento. El multiplexor MUX *F* selecciona entre la salida de la ALU y la salida del desplazador. El multiplexor MUX *D* selecciona entre la salida del multiplexor MUX *F* o el dato exterior puesto en la entrada «Data In» para ser puesto en el Bus *D*. Dicho bus se conecta a las entradas de todos los registro. Las entradas de selección de destino determinan qué registro se carga con el dato del Bus *D*. Al estar descodificadas las entradas de selección, sólo una señal de carga del registro, Load, se activa en cada transferencia de datos a los registros desde el Bus *D*. Un señal de habilitación de carga, Load Enable, fuerza a 0 todas las señales de carga, Load, utilizando una puerta AND, se utiliza para las transferencias en las que no se cambia el contenido de ninguno de los cuatros registros.

Es útil tener acceso a cierta información basada en los resultados de las operaciones de la ALU, y que esté disponible para ser utilizada por la unidad de control de la CPU para tomar decisiones. En la Figura 10-1 se muestran cuatro bits de status en la ALU. Los bits de status de acarreo, *C*, y de *overflow*, *V*, se explicaron juntamente en la Figura 5-9. El bit de status cero, *Z*, es 1 si la salida de la ALU tiene todo sus bits a cero, y es 0 en caso contrario. Así, $Z = 1$ si el resultado de una operación es cero, y $Z = 0$ si el resultado es distinto de cero. El bit de status de signo, *N*, es el bit más a la izquierda de la salida de la ALU, que es el bit de signo para el resultado en representación de número con signo. Además, también se pueden incorporar bits de status para el desplazador si se desea. La unidad de control para la ruta de datos dirige el flujo de información a través de los buses, la ALU, el desplazador y los registros mediante señales a las entradas de selección. Por ejemplo, para realizar la microoperación:

$$R1 \leftarrow R2 + R3$$

la unidad de control de la ruta de datos debe proporcionar los valores de selección binarios para el siguiente conjunto de entradas de control:

1. «*A* select», para colocar el contenido de *R*2 en el dato *A*, aquí, el Bus *A*.
2. «*B* select», para colocar el contenido de *R*3 sobre la entrada 0 del MUX *B*, y «*MB* select», para poner la entrada o del MUX *B* en el bus *B*.
3. «*G* select», para realizar la operación aritmética $A + B$.
4. «*MF* select», para colocar la salida de la ALU a la salida del MUX *F*.
5. «*MD* select», para colocar la salida del MUX *F* en el Bus *D*.



□ **FIGURA 10-1**
Diagrama de bloques de una ruta de datos genérica

6. «Destination select», para seleccionar *R*1 como destino del dato en el Bus *D*.
7. «Load enable», para habilitar un registro, en este caso, *R*1, para su carga.

El conjunto de valores se debe generar y debe estar disponible en las líneas de control correspondientes con antelación en el ciclo de reloj. Los datos en binario de los dos registros fuente se deben propagar a través de los multiplexores, la ALU, y llegar a la entrada del registro

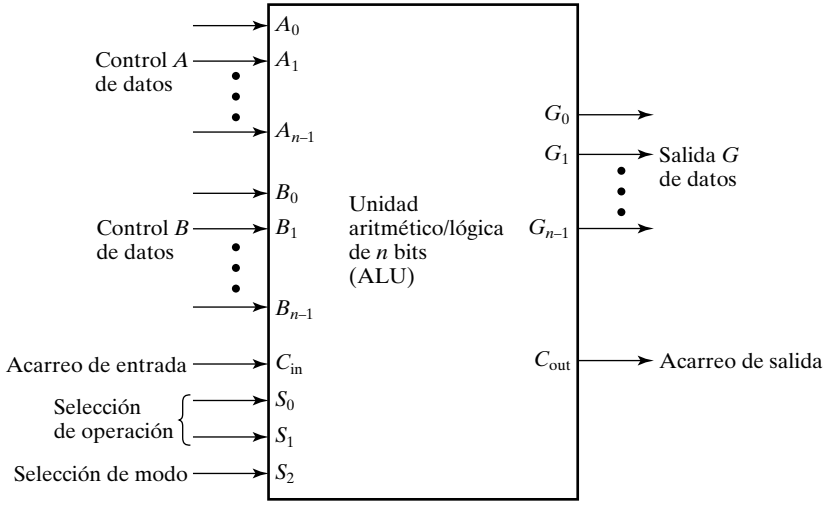
destino, todo durante el tiempo restante del mismo ciclo de reloj. Luego, cuando el siguiente flanco de subida del reloj llega, se carga el dato que está en el Bus *D* en el registro destino. Para conseguir una operación rápida, la ALU y el desplazador se construyen con lógica combinacional con un número limitado de niveles, como un sumador con acarreo anticipado.

10–3 UNIDAD ARITMÉTICO-LÓGICA

La ALU es un circuito combinacional que realiza un conjunto de microoperaciones aritméticas y lógicas básicas. La ALU tiene unas líneas de selección utilizadas para determinar la operación a realizar. Las líneas de selección son descodificadas dentro de la ALU, de forma que k líneas de selección pueden especificar hasta 2^k operaciones distintas.

La Figura 10-2 muestra el símbolo de una ALU típica de n bits. Las n entradas para el dato A se combinan con las n entradas de datos de B para generar el resultado en las salidas G. La entrada de modo de selección S_2 distingue entre operaciones aritméticas y lógicas. Las dos entradas S_1 y S_0 y la entrada de acarreo, C_{in}, especifican las ocho operaciones aritméticas con S_2 a 0. Las entradas S_0 y C_{in} especifican las cuatro operaciones lógicas con S_2 a 1.

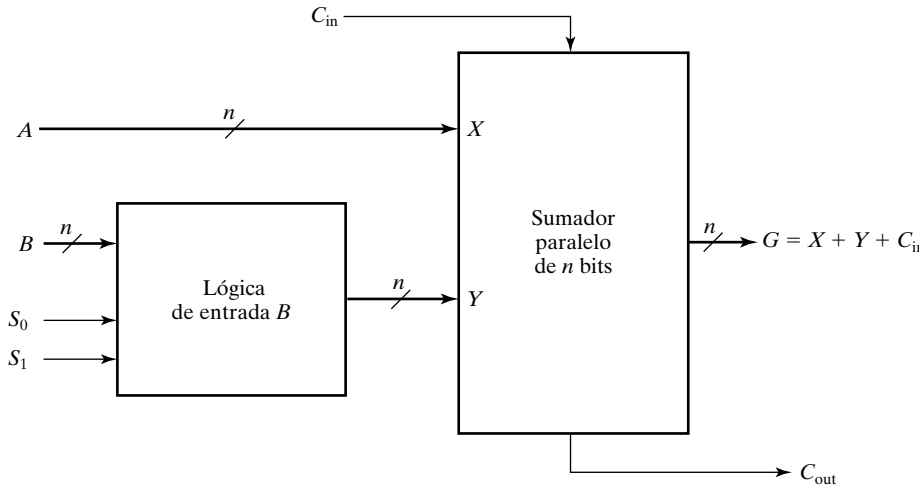
Llevaremos a cabo el diseño de esta ALU en tres etapas. Primero diseñaremos la parte aritmética, luego diseñaremos la parte lógica y, finalmente, combinaremos ambas partes para construir la ALU.



□ **FIGURA 10-2**
Símbolo para una ALU de n bits

Circuito aritmético

El componente básico de un circuito aritmético es un sumador en paralelo construido con un cierto número de sumadores completos conectados en cascada, como se mostró en la Figura 5-5. Controlando los datos de entrada al sumador en paralelo, es posible obtener diferentes tipos de operaciones aritméticas. El diagrama de bloques de la Figura 10-3 muestra una determinada configuración en la que un conjunto de entradas al sumador en paralelo se controlan mediante las líneas de selección S_1 y S_0. Es un circuito aritmético de n bits, con dos entradas A y B y una



□ **FIGURA 10-3**
Diagrama de bloques de un circuito aritmético

salida *G*. Los *n* bits de *B* pasan a través de una lógica hasta llegar a la entrada *Y* del sumador en paralelo. La entrada de acarreo, C_{in}, va a la entrada de acarreo del sumador completo de la posición menos significativa. La salida de acarreo, C_{out}, procede del sumador completo de la posición más significativa. La salida del sumador en paralelo se calcula como la suma aritmética de:

$$G = X + Y + C_{in}$$

donde *X* es el número en binario de la entrada *A* e *Y* es el número binario procedente de la salida de la lógica de entrada. C_{in} es la entrada de acarreo, que puede ser 0 o 1. Véase que el símbolo + de la ecuación denota a la suma aritmética.

La Tabla 10-1 muestra las operaciones aritméticas que se pueden realizar controlando el valor de *Y* con las dos entradas de selección S_1 y S_0. Si se ignoran las entradas de *B* y se pone a 0 todas las entradas de *Y*, la salida es $G = A + 0 + C_{in}$, es decir, $G = A$ cuando $C_{in} = 0$ y $G = A + 1$ cuando $C_{in} = 1$. En el primer caso, tenemos una transferencia directa de la entrada *A* a la salida *G*. En el segundo caso, el valor de *A* se incrementa en uno. Para efectuar una suma aritmética es necesario aplicar la entrada *B* a la entrada *Y* del sumador paralelo, es decir, $G = A + B$ cuando $C_{in} = 0$. La substracción se consigue realizando el complemento de la entra-

□ **TABLA 10-1**
Tabla de las funciones del circuito aritmético

Selección		**Entrada**	$G = A + Y + C_{in}$	
S_1	S_0	Y	$C_{in} = 0$	$C_{in} = 1$
0	0	Todo ceros	$G = A$ (transferencia)	$G = A + 1$ (incremento)
0	1	B	$G = A + B$ (suma)	$G = A + B + 1$
1	0	$\overline{B}$	$G = A + \overline{B}$	$G = A + \overline{B} + 1$ (resta)
1	1	Todo unos	$G = A - 1$ (decremento)	$G = A$ (transferencia)

da B y aplicándolo a la entrada Y del sumador en paralelo para obtener $G = A + \overline{B} + 1$ cuando $C_{in} = 1$, es decir, A más el complemento a 2 de B, que es equivalente a la substracción en complemento a 2. Todos unos es la representación en complemento a 2 de -1. De esta forma, poniendo todo los bits de la entrada Y a uno y con $C_{in} = 0$ se realiza la operación de decremento $G = A - 1$.

La lógica de entrada B de la Figura 10-3 se puede realizar con n multiplexores. Las entradas de datos para cada multiplexor para una etapa i, para $i = 0, 1, ..., n - 1$, son 0, B_i, $\overline{B}_i$ y 1, correspondientes a los valores de selección S_1S_0: 00, 01, 10 y 11, respectivamente. De esta forma, se puede construir con n sumadores completos y multiplexores de 4 a 1.

El número de puertas de la lógica B se puede reducir si, en lugar de usar multiplexores de 4 a 1, se realiza el diseño lógico de una etapa (un bit) de la lógica de entrada B. Esto se puede hacer según se muestra en la Figura 10-4(a). Las entradas son S_1, S_0 y B_i, y la salida es Y_i. Siguiendo los requerimientos especificados en la Tabla 10-1, hacemos $Y_i = 0$ cuando $S_1S_0 = 00$, y de forma similar asignado los otros tres valores restantes a Y_i para cada una de las combinaciones de las variables de selección. La salida Y_i se simplifica en el mapa de la Figura 10-4(b), resultando:

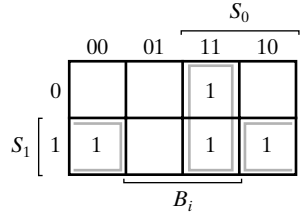
$$Y_i = B_iS_0 + \overline{B}_iS_1$$

donde S_1 y S_0 son entradas comunes a todas las n etapas: cada etapa i se asocia con la entrada B_i y la salida Y_i, para $i = 0, 1, ..., n - 1$. Esta lógica se corresponde con un multiplexor 2 a 1, con B_i en la entrada de selección y S_1 y S_0 en las entradas de datos.

La Figura 10-5 muestra el diagrama lógico de un circuito aritmético para $n = 4$. Los cuatro circuitos sumadores completos (FA) constituyen el sumador paralelo. El acarreo de la primera etapa es la entrada de acarreo, C_{in}. Los restantes acarreos se han conectado internamente desde una etapa a la siguiente. Las variables de selección son S_1, S_0 y B_i. Las variables S_1 y S_0 controlan todas las entradas Y de los sumadores completos según la función booleana que se deriva de la Figura 10-4(b). Siempre que C_{in} sea 1, $A + Y$ tiene un 1 sumado. En la Tabla 10-2 se enumeran las ocho operaciones aritméticas del circuito en función de S_1, S_0 y C_{in}. Es interesante resaltar que la operación $G = A$ aparece dos veces en la tabla. Esto es una consecuencia inofensiva de emplear C_{in} como una de las variables de control cuando se llevan a cabo las instrucciones de incremento y decremento.

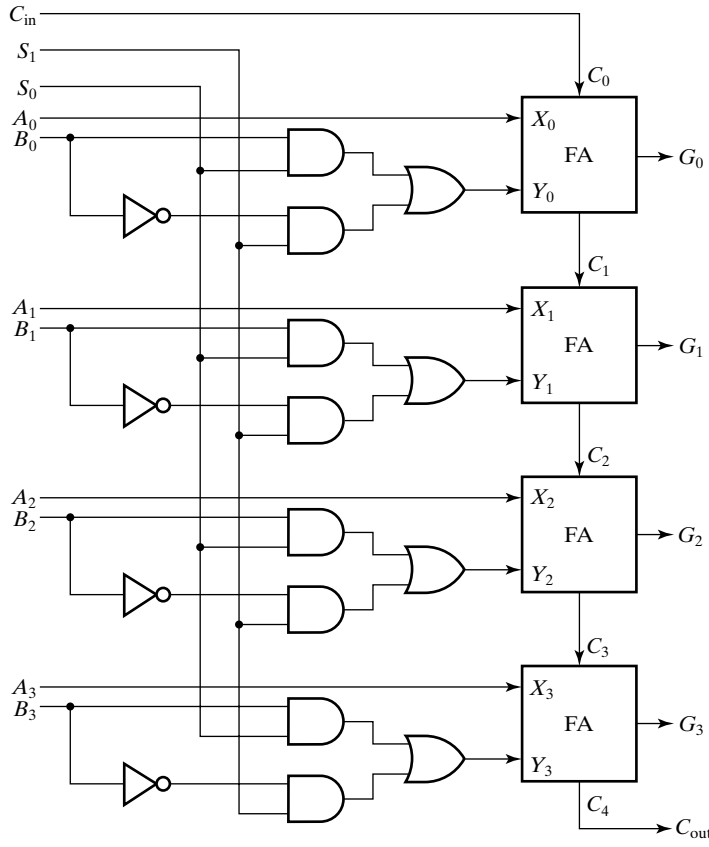
Entradas			Salidas	
S_1	S_0	B_i	Y_i	
0	0	0	0	$Y_i = 0$
0	0	1	0	
0	1	0	0	$Y_i = B_i$
0	1	1	1	
1	0	0	1	$Y_i = \overline{B}_i$
1	0	1	0	
1	1	0	1	$Y_i = 1$
1	1	1	1	

(a) Tabla de verdad



(b) Simplificación: $Y_i = B_iS_0 + \overline{B}_iS_1$

□ **FIGURA 10-4**
Etapa de la lógica de entrada B de un circuito aritmético

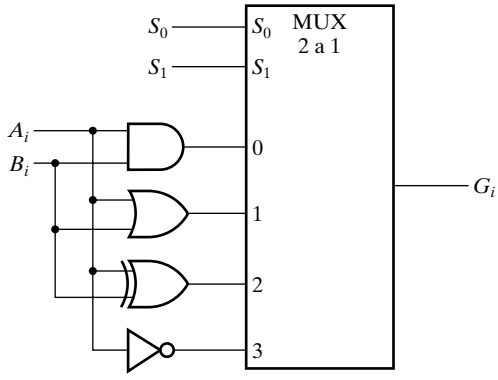


□ **FIGURA 10-5**
Diagrama lógico de un circuito lógico de 4 bits

Circuito lógico

Las microoperaciones lógicas manipulan los bits de los operando tratando cada bit de un registro como una variable binaria, dando lugar a operaciones de tipo *bitwise*. Hay cuatro operaciones comúnmente utilizadas, AND, OR, XOR y NOT, a partir de las cuales se pueden obtener otras funciones.

La Figura 10-6(a) muestra una etapa de un circuito lógico. Consiste en cuatro puertas y un multiplexor 4 a 1, aunque una simplificación podría dar lugar a una lógica menos compleja. Cada una de las cuatro operaciones se genera a través de una puerta que realiza la lógica requerida. Las salidas de las puertas se aplican a las entradas del multiplexor con las dos variables de selección S_1 y S_0. Estas escogen una de las entradas de datos del multiplexor y direcciona su valor a la salida. El diagrama muestra una etapa típica con el subíndice *i*. Para el circuito lógico de *n* bits, el diagrama debe repetirse *n* veces, para $i = 0, 1, ..., n - 1$. Las variables de selección se aplican a todas las etapas. En la tabla de funciones de la Figura 10-6(b) se enumeran las operaciones obtenidas para cada combinación de los valores de selección.



(a) Diagrama lógico

S_1	S_0	Salida	Operación
0	0	$G = A \wedge B$	AND
0	1	$G = A \vee B$	OR
1	0	$G = A \oplus B$	XOR
1	1	$G = \overline{A}$	NOT

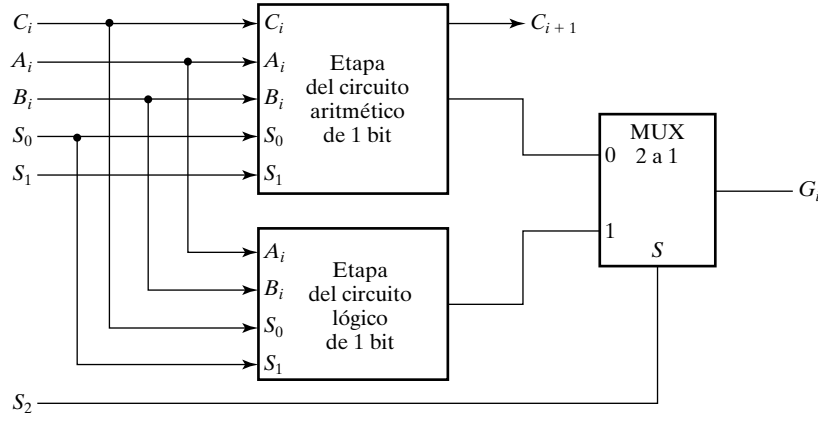
(b) Tabla de funciones

□ **FIGURA 10-6**
Etapa del circuito lógico

Unidad lógico-aritmética

El circuito lógico se puede combinar con el circuito aritmético para generar una ALU. Las variables de selección S_1 y S_0 pueden ser comunes en ambos circuitos, con tal de que usemos una tercera variable para diferenciar los dos circuitos. En la Figura 10-7 se muestra la configuración para una etapa de la ALU. Las salidas de los circuitos aritméticos y lógicos de cada etapa se conectan a un multiplexor 2 a 1 con S_2 como variable de selección. Si $S_2 = 0$, se selecciona la salida aritmética y, cuando $S_2 = 1$, se selecciona la salida lógica. Véase que el diagrama muestra sólo una etapa típica de una ALU; el circuito debe repetirse *n* veces para una ALU de *n* bits. La salida de acarreo C_{i+1} y de una determinada etapa aritmética debe conectarse a la entrada de acarreo C_i y de la siguiente etapa de la secuencia. La entrada de acarreo de la primera etapa es la entrada de acarreo C_{in}, que también actúa como variable de selección para las operaciones aritméticas.

La ALU especificada en la Figura 10-7 proporciona ocho operaciones aritméticas y cuatro lógicas. Cada operación se selecciona a través de las variables S_2, S_1, S_0 y C_{in}. En la Tabla 10-2



□ **FIGURA 10-7**
Una etapa de la ALU

□ **TABLA 10-2**
Tabla de funciones de la ALU

Selección de la operación					
S_2	S_1	S_0	C_{in}	**Operación**	**Función**
0	0	0	0	$G = A$	Transfiere A
0	0	0	1	$G = A + 1$	Incrementa A
0	0	1	0	$G = A + B$	Suma
0	0	1	1	$G = A + B + 1$	Suma con acarreo de entrada a 1
0	1	0	0	$G = A + \overline{B}$	A más el complemento a 1 de B
0	1	0	1	$G = A + \overline{B} + 1$	Resta
0	1	1	0	$G = A - 1$	Decremento de A
0	1	1	1	$G = A$	Transfiere A
1	X	0	0	$G = A \wedge B$	AND
1	X	0	1	$G = A \vee B$	OR
1	X	1	0	$G = A \oplus B$	XOR
1	X	1	1	$G = \overline{A}$	NOT (complemento a 1)

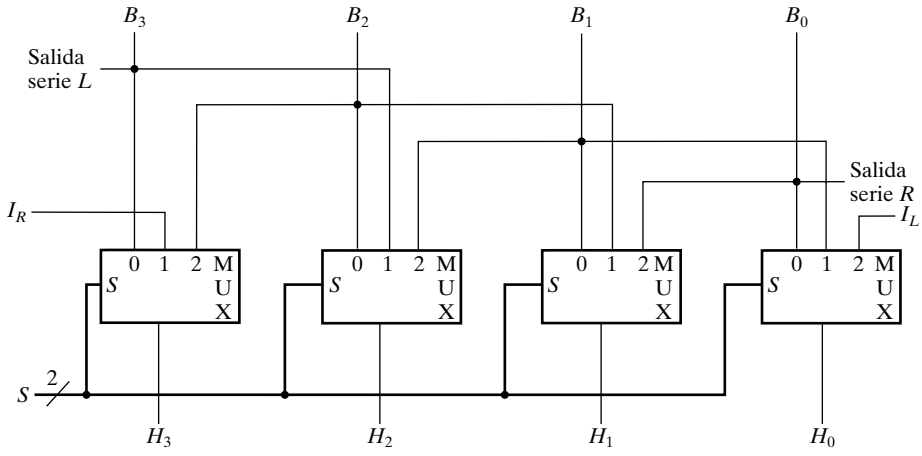
se enumeran las 12 operaciones de la ALU. Las ocho primeras son operaciones aritméticas y se seleccionan con $S_2 = 0$. Las siguientes cuatro son operaciones lógicas y se seleccionan con $S_2 = 1$. Se dan los códigos de selección usando el menor número de bits como sea posible, S_0 y C_{in} se usan para controlar la selección de las operaciones lógicas en lugar de S_2 y S_1. La entrada de selección S_1 no tiene efecto en la selección de las operaciones lógicas, por lo que se marca con una X indicando que su valor puede ser 0 o uno indistintamente. Posteriormente, en el diseño, se asigna el valor 0 para las operaciones lógicas.

La lógica de la ALU que hemos diseñado no es tan simple como se podría hacer y tiene un número bastante alto de niveles de puertas lógicas que contribuyen a elevar el retardo de propagación del circuito. Con el uso de las herramientas software de simplificación podemos simplificar esta lógica y reducir el retardo. Por ejemplo, es bastante fácil simplificar la lógica para una sola etapa de la ALU. Para un valor real de *n*, un medio posterior de reducir el retardo de propagación del acarreo de la ALU es utilizar necesariamente el sumador con acarreo anticipado presentado en la Sección 5-2.

10-4 EL DESPLAZADOR

El desplazador realiza el desplazamiento del valor presente en el Bus *B*, colocando el resultado en una entrada del MUX *F*. El desplazador básico realiza uno de los tipos de transformación sobre los datos: desplazamiento a la derecha y desplazamiento a la izquierda.

Una elección, aparentemente obvia de un desplazador, podría ser un registro bidireccional con carga en paralelo. Los datos del Bus *B* se pueden transferir al registro en paralelo y luego desplazarlo a la derecha, o a la izquierda o no desplazarlo. Un pulso de reloj carga la salida del Bus *B* en el registro de desplazamiento y en un segundo pulso de reloj se realiza el desplazamiento. Finalmente, en un tercer pulso de reloj se transfiere el dato del registro de desplazamiento al registro de destino seleccionado.



□ **FIGURA 10-8**
Desplazador básico de 4 bits

Alternativamente, la transferencia de un registro fuente a un registro destino se puede hacer en un único pulso de reloj si el desplazador se realiza con un circuito combinacional, como se hizo en el Capítulo 5. Debido a que la operación más rápida es con un pulso de reloj, en lugar de tres, se prefiere utilizar este último método. En un desplazador combinacional, las señales se propagan a través de las puertas sin necesidad de un pulso de reloj. Aquí, el único pulso necesario para un desplazamiento en la ruta de datos es para cargar el dato del Bus *H* en el registro de destino seleccionado.

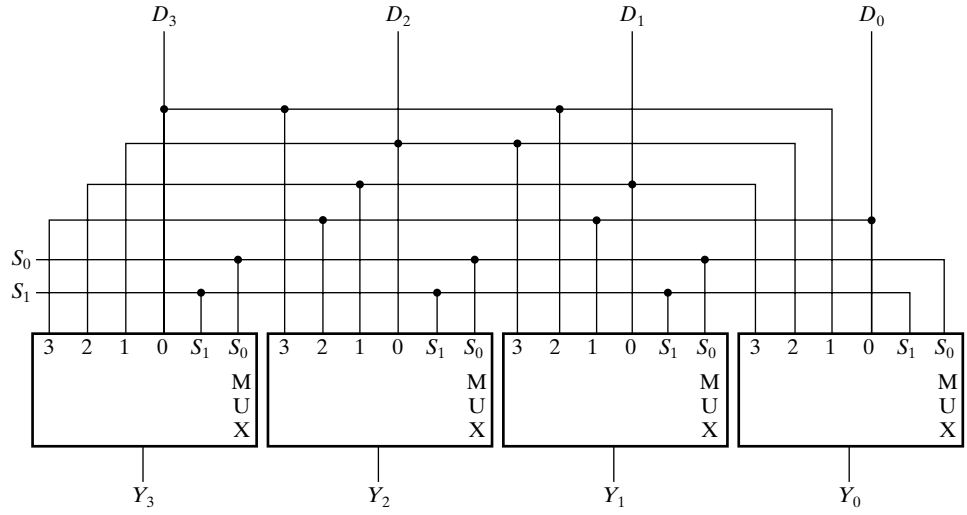
Un desplazador combinacional se puede construir con multiplexores, como se muestra en la Figura 10-8. La variable de selección *S* se aplica a los cuatro multiplexores para seleccionar el tipo de operación en el desplazador. $S = 00$ provoca que *B* pase a través del desplazador sin cambios. $S = 01$ hace una operación de desplazamiento a la derecha, y $S = 10$ produce una operación de desplazamiento a la izquierda. El desplazamiento a la derecha llena la posición en la izquierda con el valor de la entrada serie I_R. El desplazamiento a la izquierda llena la posición a la derecha con el valor en la entrada seria I_L. Las salidas están disponibles en la salida serie *R* y la salida serie *L* para los desplazamientos a la derecha e izquierda, respectivamente.

El diagrama de la Figura 10-8 muestra sólo cuatro etapas del desplazador, el cual tiene *n* etapas en un sistema con operandos de *n* bit. Se pueden añadir variables de selección para especificar que entra por I_R e I_L durante un desplazamiento de una sola posición. Nótese que para desplazar un operando $m > 1$ posiciones, el desplazador debe realizar una serie de *m* desplazamientos de una posición, requiriendo *m* ciclos de reloj.

Barrel Shifter

En las aplicaciones con rutas de datos se deben realizar con frecuencia desplazamientos de más de una posición en un solo ciclo de reloj. Un *barrel shifter*[1] es un circuito combinacional que desplaza o rota los bits del dato de entrada un cierto número de posiciones especificado mediante un valor binario colocado en un conjunto de líneas de selección. El desplazamiento que va-

[1] *N. del T.*: El término *barrel shifter* se podría traducir por desplazador de tonel pero, aparte de no decir mucho, no se suele utilizar una traducción de este término.



□ **FIGURA 10-9**
Barrel Shifter de 4 bits

mos a considerar aquí es una rotación a la izquierda, es decir, el dato en binario se desplaza a la izquierda, con los bits procedentes de la parte más significativos del registro yendo a parar a la parte menos significativa del registro.

En la Figura 10-9 se muestra una versión de cuatro bits de este tipo de registro, *barrel shifter*. Contiene cuatro multiplexores con las líneas de selección en común S_1 y S_0. Las variables de selección determinan el número de posiciones que el dato de entrada va a ser desplazado a la izquierda mediante una rotación. Si $S_1S_0 = 00$, no se efectúa desplazamiento y el dato de entrada tiene un camino directo a las salidas. Si $S_1S_0 = 01$, el dato se rota una posición, yendo D_0 a Y_1, D_1 a Y_2, D_2 a Y_3 y D_3 a Y_0. Si $S_1S_0 = 10$, la entrada se rota dos posiciones, y si $S_1S_0 = 11$ se rota tres posiciones. En la Tabla 10-3 se da la tabla de función de un *barrel shifter* de 4 bits. Para cada valor binario de las variables de selección, la tabla enumera las entradas que van a la salida correspondiente. De manera que, para rotar tres posiciones, S_1S_0 deben se igual a 11, haciendo que D_0 vaya a Y_3, D_1 a Y_0, D_2 a Y_1 y D_3 a Y_2. Véase que, usando este *barrel shifter* con rotación a la izquierda, también se puede generar cualquier rotación a la derecha que se desee. Por ejemplo, una rotación de tres posiciones a la izquierda es lo mismo que una rotación a la derecha para un *barrel shifter* de 4 bits. En general, un *barrel shifter* de 2^n bits, una rotación de i posiciones a la izquierda es lo mismo que una rotación a la derecha de $2^n - i$.

□ **TABLA 10-3**
Tabla de función para un *Barrel Shifter* de 4 bits

Selección		**Salidas**				
S_1	S_0	Y_3	Y_2	Y_1	Y_0	**Operación**
0	0	D_3	D_2	D_1	D_0	No hay rotación
0	1	D_2	D_1	D_0	D_3	Rota una posición
1	0	D_1	D_0	D_3	D_2	Rota dos posiciones
1	1	D_0	D_3	D_2	D_1	Rota tres posiciones

Un *barrel shifter* con 2^n líneas de entradas y salida necesita 2^n multiplexores, teniendo cada uno 2^n entrada de datos y n entradas de selección. El número de posiciones que el dato puede rotar se especifica por el número de variables de selección, que pueden variar en un rango entre 0 a $2^n - 1$ posiciones. Para un n grande, el fan-in de las puertas es demasiado alto, de forma que *barrel shifters* más grandes están formados por capas de multiplexores, como se muestra en la Sección 12-3, de estructuras especiales diseñadas a nivel transistor.

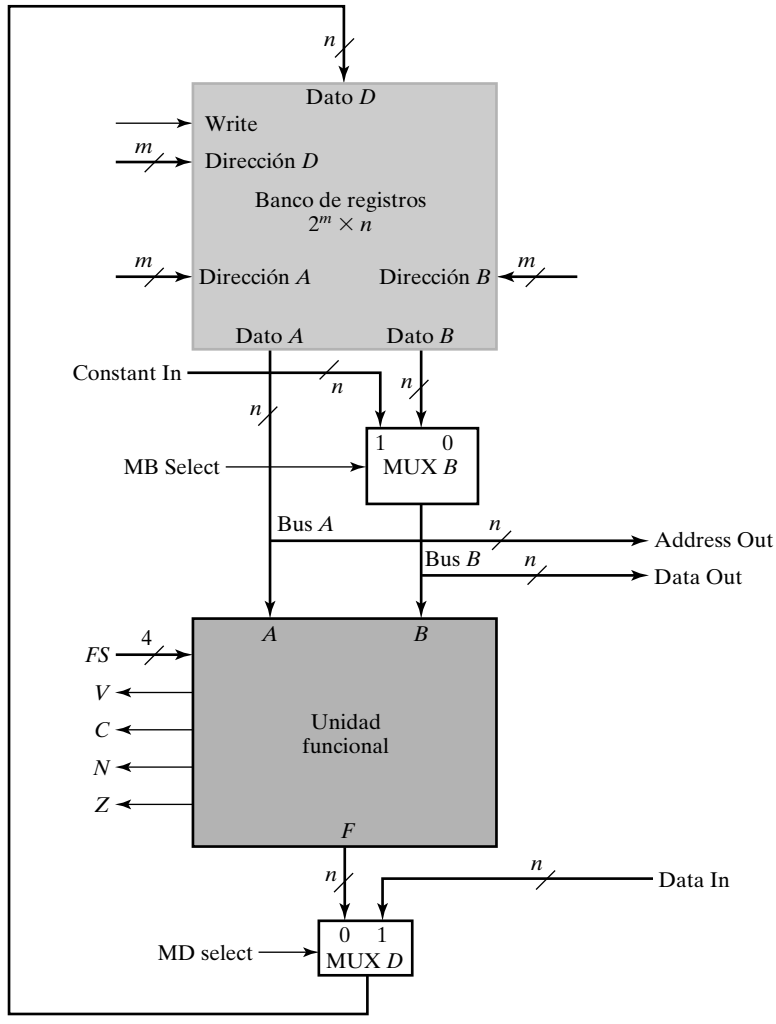
10-5 REPRESENTACIÓN DE RUTAS DE DATOS

La ruta de datos de la Figura 10-1 incluye registros, lógica de selección para los registros, la ALU, el desplazador y tres multiplexores adicionales. En una estructura jerárquica podríamos reducir la aparente complejidad de esta ruta de datos. Además, y como se ilustra en el banco de registro (en inglés *Register File*) que se presenta a continuación, el uso de una jerarquía permite el diseño donde un módulo que puede ser reemplazado por otro y, de esta forma, no estar atado a un determinado diseño de una lógica concreta.

Una ruta de datos típica tiene más de cuatro registros. De hecho, es muy frecuente encontrar procesadores con 32 o más registros. La construcción del bus de un sistema con un gran número de registros requiere técnicas diferentes. Un conjunto de registros en los se que realizan microoperaciones comunes pueden estar organizados en un banco. El banco de registros típico es un tipo especial de memoria rápida que permite leer y escribir una o más palabras simultáneamente. Funcionalmente, un banco de un solo registro contiene el equivalente a la lógica, mostrada con sombra azul, de la Figura 10-1. Debido a que los bancos de registros tienen la misma naturaleza que una memoria, las entradas de selección *A* select, *B* select y Destination select, funcionan como tres direcciones. Según se muestra en la Figura 10-1 en azul y en el símbolo del banco de registros en la Figura 10-10, la dirección *A* accede a una palabra para leerse en el bus *A* , la dirección *B* accede a una segunda palabra para leerse en el bus *B*, y la dirección *D* accede a otra palabra del bus *D* para ser escrita. Todos estos accesos tienen lugar en el mismo ciclo de reloj. También se proporciona la entrada Write, correspondiente a la señal Load Enable. Cuando está a 1, la señal Write permite que los registros sean cargados durante el presente ciclo de reloj, y cuando está a 0, evita la carga de los registros. El tamaño del banco de registros es $2^m \times n$, donde m es el número de bits de direcciones de los registros y n es el número de bits por registro. En la ruta de datos de la Figura 10-1, $m = 2$, dando lugar a cuatro registros y n está sin especificar.

Puesto que la ALU y el desplazador son unidades de proceso compartidas con salidas que se seleccionan con el MUX *F*, es conveniente agrupar las dos unidades y el MUX para formar una unidad funcional compartida. El bloque sombreado en gris de la Figura 10-1 resalta dicha unidad funcional, y se representa con el símbolo dado en la Figura 10-10. Las entradas a la unidad funcional son el Bus *A* y el Bus *B*, y la salida de la unidad va a parar al MUX *D*. La unidad funcional también tiene cuatro bits de status: *V*, *C*, *N* y *Z*, que son salidas adicionales de dicha unidad.

En la Figura 10-1 se pueden observar tres conjuntos de entradas de selección: *G* select, *H* select y *MF* select. En la Figura 10-10 hay un solo conjunto de entradas de selección etiquetadas como *FS* (del inglés *Function Select*). Para especificar completamente el símbolo de la unidad funcional de la figura, todos los códigos de *MF* select, *G* select y *H* select se deben definir en términos de códigos para *FS*. En la Tabla 10-4 se definen estas transformaciones de códigos. Los códigos para *FS* se dan en la columna izquierda. A partir de la Tabla 10-4, es evidente que *MF* es 1 para los dos bits más a la izquierda de *FS*, ambos iguales a 1. Si *F* select = 0, enton-



□ **FIGURA 10-10**
Diagrama de bloques de una ruta de datos que utiliza un banco de registros una unidad funcional

ces los códigos de *G* select determinan la función de la salida de la unidad funcional. Si *MF* select = 1, entonces los códigos de *H* select determinan la función de salida de la unidad funcional. Para mostrar esta dependencia, los códigos que determinan la función de la unidad funcional se resaltan en azul en la tabla. A partir de la Tabla 10-4 se pueden realizar las transformaciones de los códigos utilizando ecuaciones booleanas: $MF = F_3 \cdot F_2$, $G_3 = F_3$, $G_2 = F_2$, $G_1 = F_1$, $G_0 = F_0$, $H_1 = F_1$ y $H_0 = F_0$.

Suponemos que los bit de status no tienen sentido cuando se selecciona el desplazador, aunque en un sistema más complejo, los bits de status se pueden designar para reemplazar a los de la ALU siempre que se especifique una microoperación del desplazador. Véase que la forma de realizar los bits de status dependen de la forma específica que se ha usado para el circuito aritmético. Otras formas de realizar el diseño pueden no producir los mismos resultados.

□ **TABLA 10-4**
Códigos de *G* Select, *H* Select y *MF* Select definidos en términos de códigos de *FS*

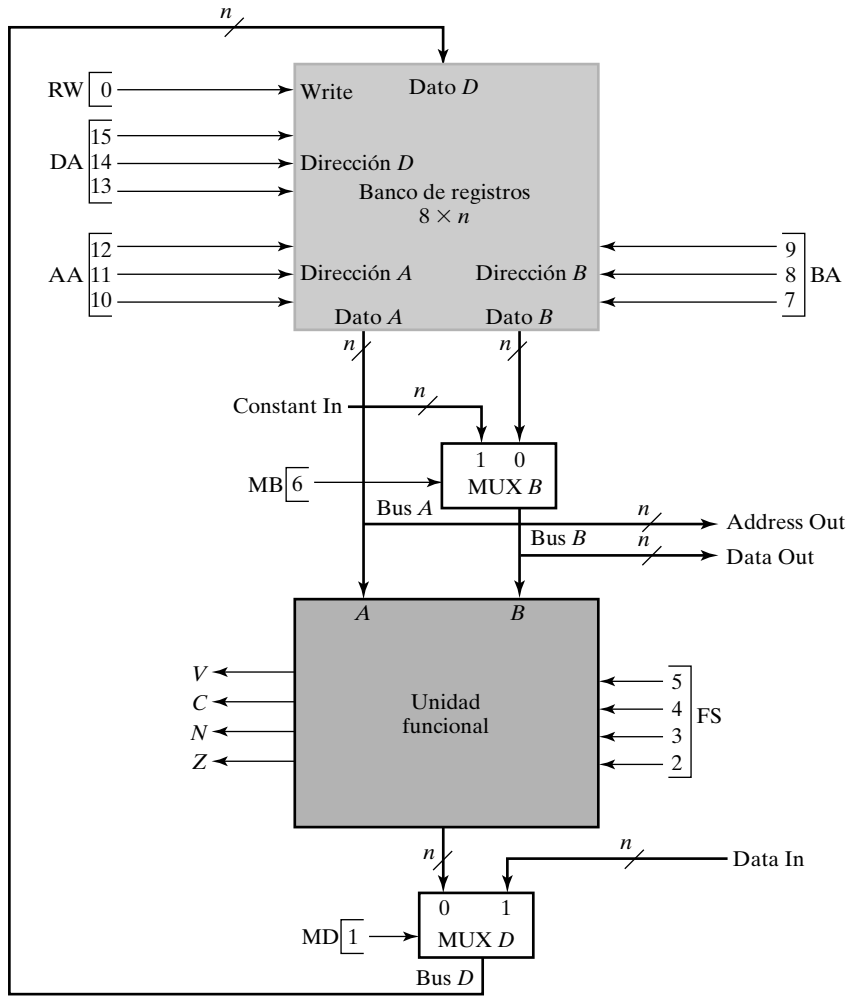
FS(3:0)	*MF* Select	*G* Select(3:0)	*H* Select(3:0)	Microoperación
0000	0	0000	XX	$F = A$
0001	0	0001	XX	$F = A + 1$
0010	0	0010	XX	$F = A + B$
0011	0	0011	XX	$F = A + B + 1$
0100	0	0100	XX	$F = A + \overline{B}$
0101	0	0101	XX	$F = A + \overline{B} + 1$
0110	0	0110	XX	$F = A - 1$
0111	0	0111	XX	$F = A$
1000	0	1X00	XX	$F = A \wedge B$
1001	0	1X01	XX	$F = A \vee B$
1010	0	1X10	XX	$F = A \oplus B$
1011	0	1X11	XX	$F = \overline{A}$
1100	1	XXXX	00	$F = B$
1101	1	XXXX	01	$F = \text{sr } B$
1110	1	XXXX	10	$F = \text{sl } B$

10-6 LA PALABRA DE CONTROL

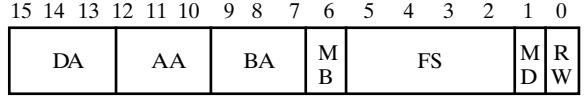
Las variables de selección de la ruta de datos controlan las microoperaciones ejecutadas dentro de éste en cualquier pulso de reloj. Para la ruta de datos de la Sección 10-5, las variables de selección controlan las direcciones para la lectura de datos del banco de registros, la función a realizar por la unidad funcional, y la carga de datos en el banco de registros, así como la selección de datos externos. Vamos a demostrar cómo estas variables de control seleccionan las microoperaciones de la ruta de datos. Se discutirá la elección de los valores de las variables de control para una microoperación típica y también se mostrará una simulación de la ruta de datos a modo ilustrativo.

En la Figura 10-11(a) se muestra una versión específica de un diagrama de bloques de la ruta de datos de la Figura 10-10. Contiene un banco de 8 registros, *R*0 a *R*7. El banco de registros proporciona las entradas a la unidad funcional mediante los Buses *A* y *B*. El multiplexor MUX *B* selecciona entre valores constantes de la entrada Constant in y los valores de registros en *B* Data. La ALU y la lógica de detección de cero dentro de la unidad funcional generan los datos binarios para los cuatro bits de status: *V* (*overflow*), *C* (acarreo), *N* (signo) y *Z* (cero). MUX *D* selecciona entre la salida de la unidad funcional y los datos en Data in como entrada del banco de registro.

Hay 16 entradas de control binarias. La combinación de sus valores especifica una palabra de control. En la Figura 10-11(b) se define la palabra de control de 16 bits. Está formada por siete partes llamadas campos, cada uno designado por un par de letras. Tres campos del registro tienen tres bits. Los campos restantes tienen un bit o cuatro bits. Los tres bits del campo DA



(a) Diagrama de bloques



(b) Palabra de control

□ **FIGURA 10-11**
Ruta de datos con variables de control

seleccionan uno de los 8 registros destino para almacenar el resultado de la microoperación. Los tres bits de AA seleccionan uno de los ocho registros fuentes para la entrada del Bus *A* a la ALU. Los tres bits de BA seleccionan un registro fuente para la entrada 0 del MUX *B*. El bit MB determina si el Bus *B* lleva el contenido del registro fuente seleccionado o un valor constante. El campo de cuatro bits FS controla la operación de la unidad de control. El campo FS contiene uno de los 15 códigos de la Tabla 10-4. El bit de MD selecciona entre la salida de la unidad funcional y el dato en Data in como entrada al Bus *D*. El último campo, RW, determina

si se escribe en un registro o no. Cuando se aplica a las entradas de control, la palabra de control de 16 bits especifica una microoperación en particular.

En la Tabla 10-5 se especifican las funciones y sus códigos de control. A cada uno de los campos se les da un código binario para cada función. El registro seleccionado por cada uno de los campos DA, AA y BA es el único con el decimal equivalente igual al número binario del código. MB selecciona tanto el registro seleccionado en el campo BA o una constante externa a la ruta de datos, Constant in. Las operaciones de la ALU, las operaciones del desplazador y la selección de las salidas de la ALU o del desplazador se especifican todas en el campo FS. El campo MD controla la información a cargarse en el banco de registros. El campo final, RW, tienen las funciones de «No escribir», evitando la escritura en cualquier registro, y «Escritura», que indica la escritura en un registro.

La palabra de control para una microoperación dada se puede derivar especificando el valor de cada campo de control. Por ejemplo, una resta dada por la sentencia

$$R1 \leftarrow R2 + \overline{R3} + 1$$

Especifica $R2$ para la entrada A de la ALU y $R3$ para la entrada B de la ALU. También especifica la operación de la unidad funcional $F = A + \bar{B} + 1$ y la selección de la salida de la unidad funcional para la entrada al banco de registros. Por último, la microoperación selecciona $R1$ como registro destino y pone RW a 1 para escribir en $R1$. La palabra para esta microinstrucción se especifica mediante sus siete campos, con los valores binarios en sus campos obtenidos de la codificación enumerada en la Tabla 10-5. La palabra binaria de control para esta microopera-

□ **TABLA 10-5**
Codificación de la palabra de control de la ruta de datos

DA, AA, BA		MB		FS		MD		RW	
Función	**Código**	**Función**	**Código**	**Función**	**Código**	**Función**	**Código**	**Función**	**Código**
$R0$	000	Registro	0	$F = A$	0000	Función	0	No escribir	0
$R1$	001	Constante	1	$F = A + 1$	0001	Data In	1	Write	1
$R2$	010			$F = A + B$	0010				
$R3$	011			$F = A + B + 1$	0011				
$R4$	100			$F = A + \bar{B}$	0100				
$R5$	101			$F = A + \bar{B} + 1$	0101				
$R6$	110			$F = A - 1$	0110				
$R7$	111			$F = A$	0111				
				$F = A \wedge B$	1000				
				$F = A \vee B$	1001				
				$F = A \oplus B$	1010				
				$F = \bar{A}$	1011				
				$F = B$	1100				
				$F = \text{sr } B$	1101				
				$F = \text{sl } B$	1110				

ción de substracción, 001_010_011_0_0101_0_1, (usamos el signo «_» por conveniencia para separar los campos) se obtiene como sigue:

Campo:	DA	AA	BA	MB	FS	MD	RW
Simbólico:	*R*1	*R*2	*R*3	Registro	$F = A + \bar{B} + 1$	Función	Escribe
Binario:	001	010	011	0	0101	0	1

La palabra de control para la microoperación y aquellas otras microoperaciones se dan en la Tabla 10-6 usando notación simbólica, y en la Tabla 10-7 usando códigos binarios.

El segundo ejemplo de la Tabla 10-6 es una operación de desplazamiento dada por la sentencia

$$R4 \leftarrow \text{sl } R6$$

Esta sentencia especifica un desplazamiento a la izquierda para el desplazador. El contenido del registro *R*6, desplazado a la izquierda, se transfiere al registro *R*4. Tenga en cuenta que, debido a que el desplazador se maneja mediante el bus *B*, la fuente para el desplazamiento se especifica en el campo BA en lugar del campo AA. Teniendo en cuenta los símbolos de cada campo, la palabra de control en binario se extrae según se muestra en la Tabla 10-7. En algunas microoperaciones no se utiliza ni el dato *A* ni el *B* del banco de registros. En estos casos, el símbolo del campo correspondiente se marca con un guión. Puesto que estos valores están sin especificar, los valores correspondientes de la Tabla 10-7 son X. Continuando con los tres últimos ejemplos de la Tabla 10-6, para tener disponibles los contenidos de los registros para un destino externo colocaremos los contenidos del registro en la salida de datos *B* del banco de registro, con RW = No escribe (0) para evitar que el banco de registros sean escritos. Para colocar una constante de poco valor en un registro o utilizar una constante de poco valor como uno de los operandos, colocaremos la constante en la entrada *Constant in*, actualizando MB para seleccionar la constante, y pasar el valor del Bus *B* a través de la ALU y el Bus *D* hasta el registro destino. Para poner a 0 el registro, el Bus *D* se pone todo a 0s usando el mismo registro tanto para el Bus de datos *A* y el Bus de datos *D* con la operación XOR especificada (FS = 1010) y MD = 0. El campo DA se actualiza con el código del registro destino y RW está con el valor Escribe (1).

Es evidente, a partir de estos ejemplos que, algunas microoperaciones se pueden realizar mediante la misma ruta de datos. Se pueden realizar secuencias de dichas microoperaciones utilizando una unidad de control que produzca la secuencia apropiada de palabras de control.

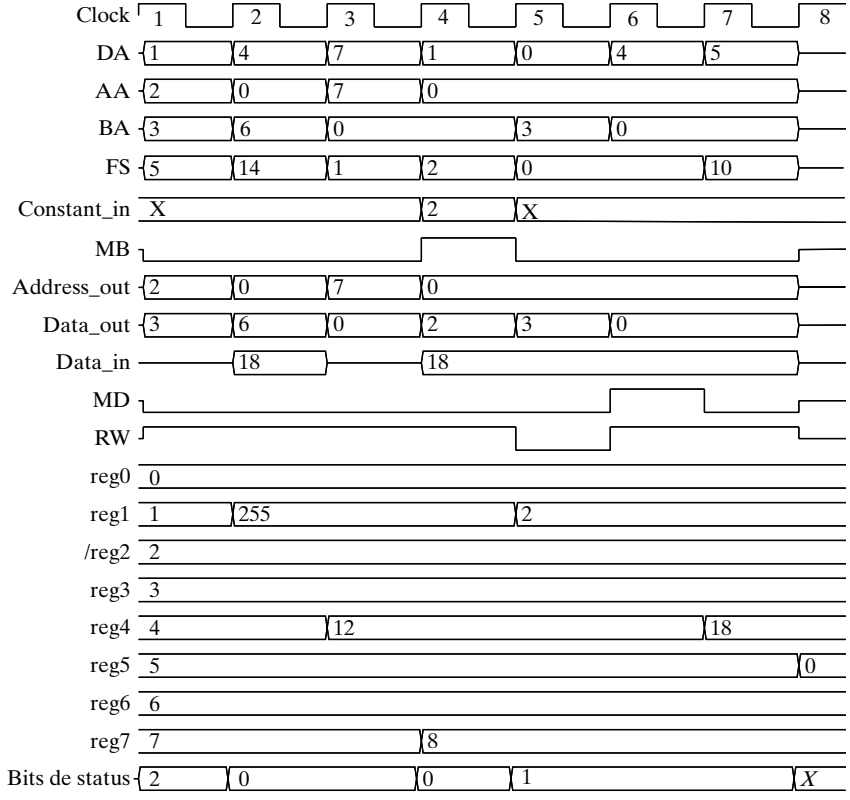
□ **TABLA 10-6**
Ejemplo de microoperaciones para la ruta de datos utilizando notación simbólica

Microoperación	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	*R*1	*R*2	*R*3	Registro	$F = A + \bar{B} + 1$	Función	Escribe
$R4 \leftarrow \text{sl } R6$	*R*4	—	*R*6	Registro	$F = \text{sl } B$	Función	Escribe
$R7 \leftarrow R7 + 1$	*R*7	*R*7	—	Registro	$F = A + 1$	Función	Escribe
$R1 \leftarrow R0 + 2$	*R*1	*R*0	—	Constante	$F = A + B$	Función	Escribe
Data out $\leftarrow R3$	—	—	*R*3	Registro	—	—	No Escribe
$R4 \leftarrow$ Data in	*R*4	—	—	—	—	Data in	Escribe
$R5 \leftarrow 0$	*R*5	*R*0	*R*0	Registro	$F = A \oplus B$	Función	Escribe

□ **TABLA 10-7**
Ejemplo de microoperaciones de la Tabla 10-6 utilizando palabras de control

Microoperación	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	001	010	011	0	0101	0	1
$R4 \leftarrow$ sl $R6$	100	XXX	110	0	1110	0	1
$R7 \leftarrow R7 + 1$	111	111	XXX	0	0001	0	1
$R1 \leftarrow R0 + 2$	001	000	XXX	1	0010	0	1
Data out $\leftarrow R3$	XXX	XXX	011	0	XXX	X	0
$R4 \leftarrow$ Data in	100	XXX	XXX	X	XXX	1	1
$R5 \leftarrow 0$	101	000	000	0	1010	0	1

Para completar esta sección, realizaremos una simulación de la ruta de datos de la Figura 10-11. El número de bits de cada registro, *n*, es igual a 8. Se utiliza una representación decimal sin signo, por ser la más conveniente para leer el resultado de la simulación, para todas las señales con varios bits. Suponemos que las microoperaciones de la Tabla 10-7, se ejecutan secuencialmente, dando las entradas a la ruta de datos, y el contenido inicial de cada registro es su número en decimal (por ejemplo $R5$ contiene $0000\ 0101_2 = (5)_{10}$). La Figura 10-12 da el resultado de dicha simulación. El primer valor dibujado es el reloj con sus ciclos numerados para



□ **FIGURA 10-12**
Simulación de una secuencia de microoperaciones de la Tabla 10-7

facilitar su referencia. Las entradas, salidas y el estado de la ruta de datos se dan aproximadamente en el orden del flujo de información a través de la ruta. Las primeras cuatro entradas son los campos de la palabra de control principal, que especifican las direcciones del registro que determina las salidas del banco de registros, y la selección de la función. A continuación están las entradas *Constant in* y MB, que controlan la entrada al Bus *B*. Seguidamente están las salidas Address out y Data out, que son las salidas al Bus *A* y *B*, respectivamente. Las tres variables siguientes: Data in, MD y RW, son las últimas entradas a la ruta de datos. A continuación de éstas aparece el contenido de los ochos registros y los bits de status, que se dan como un vector (*V*, *C*, *N*, *Z*). El valor inicial del contenido de cada registro es su número en decimal. El valor 2 se aplica a *Constant in* solo en el ciclo 4 donde MB es igual a 1. De otro modo, el valor en Constant in que es desconocido se indica con una X. Finalmente, Data in tiene el valor 18. En la simulación, este valor llega de una memoria que se direcciona mediante Address out y tiene el valor 18 en la posición 0 con el resto de valores sin especificar en el resto de posiciones. El valor resultante, excepto cuando la dirección en Address out es 0, se representa por una línea a media altura entre el 0 y el 1, indicando que el valor es desconocido.

Resaltar en los resultados de la simulación que, los cambios en los registros, como resultado de una operación en particular, aparecen en el ciclo de reloj posterior en el que se especifica la microoperación. Por ejemplo, los resultados de la sustracción especificada en el ciclo de reloj 1 aparecen en el registro *R*1 en el ciclo de reloj 2. Esto es debido a que el resultado se carga en los flip-flops en el flanco de subida de reloj al final del ciclo de reloj 1. Por otro lado, los valores de los bits de status, Address out y Data out aparecen en el mismo ciclo de reloj según la microoperación los controla, puesto que no dependen de cuándo ocurre el flanco de subida de reloj. Como no se especifican los retardos de la lógica combinacional en la simulación, estos valores cambian al mismo tiempo que los valores de los registros. Para terminar, véase que los ocho ciclos de reloj de la simulación se usan para realizar siete microoperaciones de forma que se pueden observar los valores en los registros que resultan de la última microoperación ejecutada. Aunque los bits de status aparecen en todas las microoperaciones, no siempre tienen sentido. Por ejemplo, para las microoperaciones, $R3 =$ Data out y $R4 \leftarrow$ Data in, en los ciclos de reloj 5 y 6, respectivamente, el valor de los bits de status no están relacionados con el resultado ya que la unidad funcional no se usa en estas operaciones. Finalmente, para $R5 \leftarrow R0 \oplus R0$ en el ciclo de reloj 7, la unidad aritmética no se utiliza, por eso los valores de *V* y *C* de la unidad son irrelevantes, aunque los valores de *N* y *Z* representan el status del resultado con un entero con signo en complemento a 2.

10-7 ARQUITECTURA DE UN SENCILLO PROCESADOR

Presentamos la arquitectura de un sencillo procesador para obtener una primera comprensión del diseño de procesadores e ilustrar diseños del control en sistemas programables. En un sistema programable, una parte de la entrada al procesador consiste en una secuencia de instrucciones. Cada instrucción especifica la operación que se va a realizar en el sistema, qué operandos utiliza la operación, dónde colocar los resultados de la operación y, en algunos casos, qué instrucción se ejecuta a continuación. En los sistemas programables, las instrucciones se almacenan habitualmente en memoria, que puede ser RAM o ROM. Para ejecutar las instrucciones en secuencia, es necesario proporcionar la dirección de memoria de la instrucción a ser ejecutada. En un procesador, esta dirección procede de un registro llamado contador de programa (*PC*, del término inglés *Program Counter*). Como su nombre implica, el *PC* tiene una lógica que le permite contar. Además, para cambiar la secuencia de operaciones usando decisiones basadas en la

información de status, el *PC* necesita la capacidad de carga en paralelo. De esta manera, en el caso de un sistema programable, la unidad de control contiene un *PC* y su lógica de decisión asociada, así como la lógica necesaria para interpretar la instrucción en curso para ejecutarla. Ejecutar una instrucción significa activar la secuencia necesaria de microoperaciones en la ruta de datos (y en otras partes) necesarias para realizar la operación especificada por la instrucción. En contraste con lo anterior, nótese que, en un sistema no programable, la unidad de control no es responsable de obtener las instrucciones de la memoria, ni es responsable de la secuenciación de la ejecución de estas instrucciones. No hay *PC* ni registro similar en dichos sistemas. En su lugar, la unidad de control determina las operaciones a realizar y su secuencia, basándose solo en sus entradas y los bits de status.

Demostramos cómo las operaciones especificadas por las instrucciones de un sencillo procesador se pueden realizar mediante microoperaciones en la ruta de datos, más movimiento de información entre la ruta de datos y la memoria. También mostramos dos estructuras de control diferentes para realizar las secuencias de operaciones necesarias para controlar la ejecución del programa. El propósito aquí es ilustrar los dos métodos diferentes para el diseño de control y los efectos que tales métodos tienen en el diseño de la ruta de datos y el rendimiento del sistema. Un estudio más extensivo de los conceptos asociados con los conjuntos de instrucciones para procesadores digitales se presenta con detalle en el siguiente capítulo y se consideran diseños de CPU más complejos en el Capítulo 12.

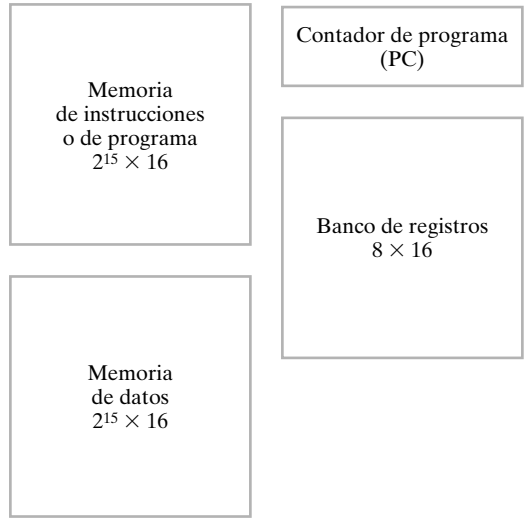
Arquitectura de conjunto de instrucciones

El usuario especifica las operaciones a llevar a cabo y su secuencia mediante un programa, que es una lista de instrucciones que especifican las operaciones, los operandos y la secuencia en la que ocurre el procesamiento. El procesado de datos desarrollado por un procesador se puede alterar especificando un nuevo programa con diferentes instrucciones o especificando las mismas instrucciones pero con datos diferentes. Las instrucciones y los datos se almacenan habitualmente juntos en la misma memoria. Según las técnicas que se discuten en el Capítulo 12, puede parecer que instrucciones y datos proceden de memorias diferentes. La unidad de control lee una instrucción de la memoria, la descodifica y la ejecuta usando una secuencia de una o más microoperaciones. La habilidad para ejecutar un programa de la memoria es la propiedad más importante de un procesador de propósito general. La ejecución de un programa de la memoria es muy distinta al de la unidad de control del multiplicador no programable, considerado anteriormente, que ejecuta una sola operación fija.

Una *instrucción* es una colección de bits que instruye al procesador para realizar una operación específica. Llamamos *colección de instrucciones* de un procesador a su conjunto de instrucciones, una descripción completa del conjunto de instrucciones a su *arquitectura de conjunto de instrucciones* (ISA, del inglés *Instruction Set Architecture*). Las arquitecturas con un conjunto sencillo de instrucciones tienen tres componentes principales: los recursos de almacenamiento, los formatos de la instrucción y las especificaciones de la instrucción.

Recursos de almacenamiento

Los recursos de almacenamiento para un procesador sencillo se representan en el diagrama de la Figura 10-13. El diagrama esboza la estructura de un procesador, según se ve por un usuario, que los programa en un lenguaje que especifica directamente la instrucción a ejecutar. Se dan los recursos que el usuario ve disponible para el almacenamiento de la información. Véase que



□ **FIGURA 10-13**
Diagrama de recursos de almacenamiento de un procesador sencillo

la arquitectura incluye dos memorias, una para almacenamiento de instrucciones y la otra para el almacenamiento de datos. Esto puede hacerse en diferentes memorias o puede hacerse en la misma memoria pero vista como si fuesen diferentes desde el punto de vista de la CPU, como se discute en el Capítulo 12. También es visible en el diagrama para el programador, un banco de registros con 8 registros de 16 bits y un contador de programa de 16 bits.

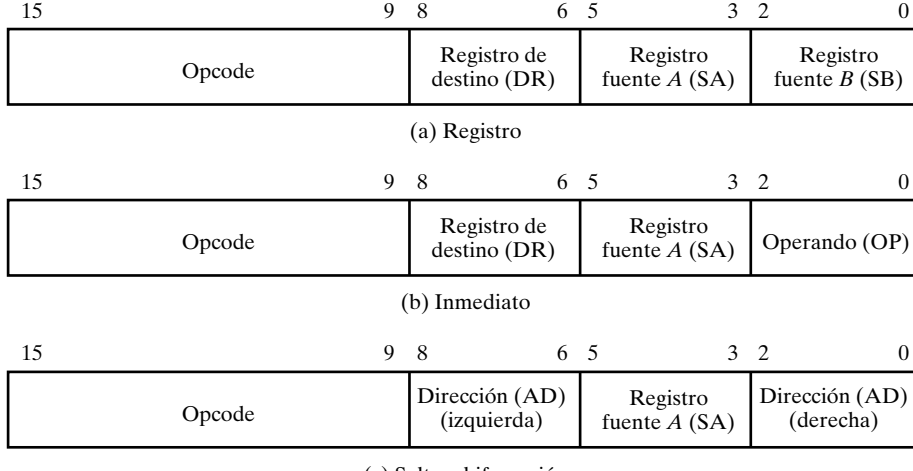
Formatos de la instrucción

El formato de una instrucción se suele describir mediante una caja rectangular que simboliza los bits de la instrucción, como aparecen en las palabras de la memoria o en un registro de control. los bits se dividen en grupos o partes llamadas *campos*. Cada campo se asigna a un elemento específico, como es el código de operaciones, un valor constante o una dirección de un banco de registros. Los diversos campos especifican diferentes funciones para la instrucción y, cuando se muestran juntos, constituyen el formato de una instrucción.

El *código de operación* de una instrucción, abreviado frecuentemente como «opcode», es un grupo de bits en la instrucción que especifica una operación, como la suma, la resta, el desplazamiento o el complemento. El número de bits necesarios para el opcode de una instrucción es función del número total de operaciones del conjunto de instrucciones. Debe estar formado, al menos, por m bits para un total de hasta 2^m operaciones distintas. El diseñador asigna una combinación de bits (un código) para cada operación. El procesador se diseña para que acepte esta configuración de bits en el momento adecuado dentro de la secuencia de actividades y proporciona la adecuada secuencia de palabras de control para ejecutar la operación especificada. Consideremos como ejemplo un procesador con un máximo de 128 operaciones distintas, entre las que se incluye la operación de suma. El código de operación asignado a esta operación está formado por 7 bits, 0000010. Cuando la unidad de control detecta este código de operación, se aplica una secuencia de palabras de control a la ruta de datos para realizar la suma que se pretende. El código de operación de una instrucción especifica la operación que se va a realizar. La operación se debe llevar a cabo usando los datos almacenados en los registros del procesador o

en la memoria (es decir, en los recursos de almacenamiento). Una instrucción, por tanto, debe especificar no sólo la operación, sino también los registros o las palabras de memoria en la que se pueden encontrar los operandos y dónde se debe colocar el resultado. Los operandos se pueden especificar en una instrucción de dos formas. Se dice que un operando se especifica explícitamente si la instrucción contiene bits especiales para su identificación. Por ejemplo, la instrucción que realiza una suma puede contener tres números binarios que especifican los registros que contienen los dos operandos y el registro que recibe el resultado. Se dice que un operando se define implícitamente si se incluye como parte de la definición de la operación en sí misma, estando representado en el código de operación en lugar de estarlo en la instrucción. Por ejemplo, en una operación de Incremento de Registro, uno de los operandos es implícitamente $+1$.

En la Figura 10-14 se ilustran tres formatos de instrucción para un procesador sencillo. Supongamos que el procesador tiene un banco con ocho registros, *R*0 a *R*7. El formato de la instrucción de la Figura 10-14(a) está compuesta por un código de operación que especifica el uso de hasta tres registros, según sea necesario. Uno de los registros se asigna como destino para el resultado y dos de los registros fuentes para los operandos. Por conveniencia, los nombres de los campos se han abreviado como: DR, para el Registro de Destino (del inglés *Destination Register*), SA para el Registro Fuente *A* (*Source Register A*) y SB para el Registro Fuente *B* (*Source Register B*). El número de campos para los registros y los registros realmente usados se determinan mediante un código de operación específico. El código de operación también especifica el uso de los registros. Por ejemplo, para una operación de substracción, supongamos que los tres bits en SA son 010, especificando a *R*2, los tres bits de SB son 011, designando a *R*3 y los tres bits de DR son 001, especificando a *R*1. Luego, el contenido de *R*3 se restará al contenido de *R*2, y el resultado se guardará en *R*1. Como ejemplo adicional, supongamos una operación de almacenamiento en la memoria. Suponga además que los tres bits de SA designan a *R*4 y que los tres bits de SB especifican a *R*5. Para esta operación en particular se supone que el registro especificado en SA contiene la dirección donde debe ser almacenado el operando y SB contiene el operando que se va a almacenar. De esta forma, el valor en *R*5 se almacena en la posición de memoria dada por el valor del registro *R*4. El campo DR no tiene ningún efecto puesto que la operación de almacenamiento evita que se escriba en el banco de registros.



□ **FIGURA 10-14**
Formato de tres instrucciones

El formato de la instrucción de la Figura 10-14(b) tiene un código de operación, dos campos para registros y un operando. El operando es una constante que se llama operando inmediato ya que está inmediatamente disponible en la instrucción. Por ejemplo, para una operación de suma inmediata, donde SA especifica *R*7, DR índica *R*2 y como operando OP es igual a 011, el valor 3 se suma al contenido de *R*7, y el resultado se la suma se coloca en *R*2. Puesto que el operando es de sólo tres bits en lugar de uno de 16 bits, el resto de los 13 bits se deben rellenar con cero o hacer una extensión de signo, como se indicó en el Capítulo 5. En esta ISA, se especifica el relleno con ceros del operando.

El formato de la instrucción de la Figura 10-14(c), en comparación con los otros dos formatos, no cambia ningún registro del banco ni el contenido de la memoria. En su lugar, afecta al orden de acceso de las instrucciones de la memoria. La posición de una instrucción a la que se va a acceder se determina mediante el contador de programa, *PC*. Normalmente, el contador de programa accede a las instrucciones en direcciones consecutivas de la memoria según el programa se ejecuta. Pero gran parte de la potencia de un procesador procede de su capacidad de cambiar el orden de ejecución a partir de los resultados que surgen a lo largo de la ejecución de las instrucciones. Estos cambios en el orden de ejecución de las instrucciones se basan en el uso de instrucciones llamadas de salto y de bifurcación.

El ejemplo de formato dado en la Figura 10-14(c) es para instrucciones de salto y bifurcación, y tienen un código de operación, un campo para registros, SA, y un campo de dirección AD. Si ocurre una bifurcación (basada posiblemente en el contenido del registro especificado), la nueva dirección se forma sumando el contenido actual del *PC* y el contenido del campo de direcciones de 6 bits. Este método de direccionamiento se le llama relativo al Contador de Programa y el campo de direcciones de 6 bits, al que se llama *dirección relativa*, se trata como un número con signo en complemento a dos. Para conservar la representación en complemento a dos, se aplica la extensión de signo a la dirección de 6 bits para formar un desplazamienmto de 16 bits antes de realizar la suma. Si el bit más a la izquierda del campo de direcciones, AD, es 1, los 10 bits a su izquierda se rellenan con unos para hacer el complemento a dos del desplazamiento. Si el bit más a la izquierda del campo de direcciones es 0, los restantes 10 bits a su izquierda se rellenan con ceros para dar un desplazamiento positivo en complemento a dos. El desplazamiento resultante se suma al contenido del *PC* para formar la dirección de la siguiente instrucción a la que se va a acceder. Por ejemplo, con el valor de *PC* igual a 55, suponemos que ocurre una bifurcación en la posición 35 si el contenido de *R*6 es igual a cero. El código de operación especificaría una bifurcación sobre la condición de cero, SA debería especificar a *R*6 y AD debería tener la representación en complemento a dos del número -20 en 6 bits. Si el contenido de *R*6 es cero, el contenido del *PC* pasará a ser $55 + (-20) = 35$, y la siguiente instrucción se accedería de la dirección 35. Por otra parte, si el contenido de *R*6 es distinto de cero, el *PC* contará ascendentemente a 56 y se accederá a la instrucción de esta dirección. Este método de direccionamiento sólo proporciona una dirección de bifurcación dentro de un pequeño intervalo por debajo y por encima del valor del *PC*. El salto proporciona un rango más amplio de direcciones usando el contenido sin signo de un registro de 16 bits como dirección de salto.

Los tres formatos de la Figura 10-14 utilizados por este sencillo procesador se estudian en este capítulo. En el Capítulo 11 se presentan y estudian otros tipos y formatos de instrucciones más generales.

Especificación de las instrucciones

La especificación de las instrucciones describe cada una de las distintas instrucciones que se pueden ejecutar en el sistema. Para cada instrucción se da el código de operación mediante un

nombre abreviado, llamado *mnemónico*, que puede usarse como representación simbólica del código de operación. Este mnemónico, junto con la representación de los campos adicionales de la instrucción del formato de la instrucción, representa la notación a utilizar en la especificación de todos los campos de la instrucción simbólicamente. Esta representación simbólica se convierte posteriormente en una representación binaria de la instrucción mediante un programa llamado *ensamblador*. Se da una descripción de la operación realizada por la instrucción, incluyendo los bits de status involucrados por dicha instrucción. Esta descripción puede estar en un texto o como notación de transferencia de registros. En la Tabla 10-8 se dan las especificaciones de las instrucciones para el procesador. Se usa, además, la notación de transferencia de registros, introducida en los capítulos anteriores, para describir la operación a realizar, y se indican los bits de status que son válidos para cada instrucción. Con el fin de ilustrar las instrucciones, supongamos que tenemos una memoria de 16 bits por palabra, con instrucciones que tienen uno de los formatos mostrados en la Figura 10-14. Las instrucciones y los datos, en binario, se colocan en la memoria como se muestra en la Tabla 10-9. Esta información almacenada representa las cuatro instrucciones que ilustran los distintos formatos. En la dirección 25 tenemos una instrucción con formato de registro, que especifica una operación que resta *R*3 de *R*2 y carga la

□ **TABLA 10-8**
Especificación de la instrucciones del procesador

Instrucción	Opcode	Mnemónico	Formato	Descripción	Bits de status
Mueve A	0000000	MOVA	RD, RA	R[DR] ← R[SA]	N, Z
Incrementa	0000001	INC	RD, RA	R[DR] ← R[SA] + 1	N, Z
Suma	0000010	ADD	RD, RA, RB	R[DR] ← R[SA] + R[SB]	N, Z
Substracción	0000101	SUB	RD, RA, RB	R[DR] ← R[SA] − R[SB]	N, Z
Decremento	0000110	DEC	RD, RA	R[DR] ← R[SA] − 1	N, Z
AND	0001000	AND	RD, RA, RB	R[DR] ← R[SA] ∧ R[SB]	N, Z
OR	0001001	OR	RD, RA, RB	R[DR] ← R[SA] ∨ R[SB]	N, Z
OR Exclusiva	0001010	XOR	RD, RA, RB	R[DR] ← R[SA] ⊕ R[SB]	N, Z
NOT	0001011	NOT	RD, RA	R[DR] ← $\overline{R[SA]}$	N, Z
Mueve B	0001100	MOVB	RD, RB	R[DR] ← R[SB]	
Desplazamiento a la derecha	0001101	SHR	RD, RB	R[DR] ← sr R[SB]	
Desplazamiento a la izquierda	0001110	SHL	RD, RB	R[DR] ← sl R[SB]	
Carga inmediata	1001100	LDI	RD, OP	R[DR] ← zf OP	
Suma inmediata	1000010	ADI	RD, RA, OP	R[DR] ← R[SA] + zf OP	
Carga	0010000	LD	RD, RA	R[DR] ← M[SA]	
Almacena	0100000	ST	RA, RB	M[SA] ← R[SB]	
Bifurcación sobre cero	1100000	BRZ	RA, AD	if (R[SA] = 0) PC ← PC + se AD	
Bifurcación sobre negativo	1100001	BRN	RA, AD	if (R[SA] < 0) PC ← PC + se AD	
Salto	1110000	JMP	RA	PC ← R[SA]	

diferencia en *R*1. Esta operación se representa simbólicamente en la columna más a la derecha de la Tabla 10-9. Véase que el código de operación de 7 bits para la resta es 0000101, o en decimal, 5. El resto de los bits de la instrucción especifican los tres registros: 001 especifica a *R*1 como registro destino, 010 especifica a *R*2 como registro fuente *A*, y 011 especifica a *R*3 como registro fuente *B*.

En la posición de memoria 35 hay una instrucción con formato de registro para almacenar el contenido de *R*5 en la posición de memoria especificada por *R*4. El código de operación es 0100000, o 32 en decimal, y se da la operación simbólicamente en la columna más a la derecha de la figura. Supongamos que *R*4 contiene el valor 70 y *R*5 el valor 80. La ejecución de esta instrucción almacenará el valor 80 en la posición 70 de la memoria, reemplazando el valor originalmente almacenado, en este caso 192.

En la dirección 45 aparece una instrucción con formato inmediato que suma 3 al contenido de *R*7 y carga el resultado en *R*2. El código de operación es 66 y el operando a sumar es 3 (011) y está en el campo OP, que son los tres últimos bits de la instrucción.

En la posición 55 aparece una instrucción de bifurcación, como se describió anteriormente. El código de operación para esta instrucción es 96, y el registro fuente *A* especificado es el *R*6. Véase que AD (izquierda) contiene 101 y AD (derecha) contiene 100. Colocando estos dos juntos, y aplicando la extensión de signo obtenemos 1111111111101100, que representa el valor −20 en complemento a dos. Si el registro *R*6 es cero, el valor −20 se suma al contenido del

□ **TABLA 10-9**
Representación de las instrucciones y datos de la memoria

Dirección en decimal	Contenido de la memoria	Opcode en decimal	Otros campos	Operación
25	0000101 001 010 011	5 (resta)	DR:1, SA:2, SB:3	$R1 \leftarrow R2 - R3$
35	0100000 000 100 101	32 (almacena)	SA:4, SB:5	$M[R4] \leftarrow R5$
45	1000010 010 111 011	66 (suma inmediata)	DR:2, SA:7, OP:3	$R2 \leftarrow R7 + 3$
55	1100000 101 110 100	96 (bifurcación sobre cero)	AD: 44, SA:6	If $R6 = 0$, $PC \leftarrow PC - 20$
70	00000000011000000	Dato = 192. Después de la ejecución de la instrucción en la posición 35, Dato = 80.		

PC dando como resultado 35. Si el contenido del registro *R*6 es distinto de cero, el nuevo valor que tomará *PC* será 56. Debemos resaltar que hemos supuesto que la suma del contenido del *PC* se efectúa antes de que el *PC* se haya incrementado, como debe ser en este sencillo procesador. En los sistemas reales, no obstante, a veces el *PC* se ha incrementado para apuntar la siguiente instrucción de la memoria. En tal caso, es necesario ajustar el valor almacenado en AD adecuadamente para obtener la dirección de bifurcación correcta.

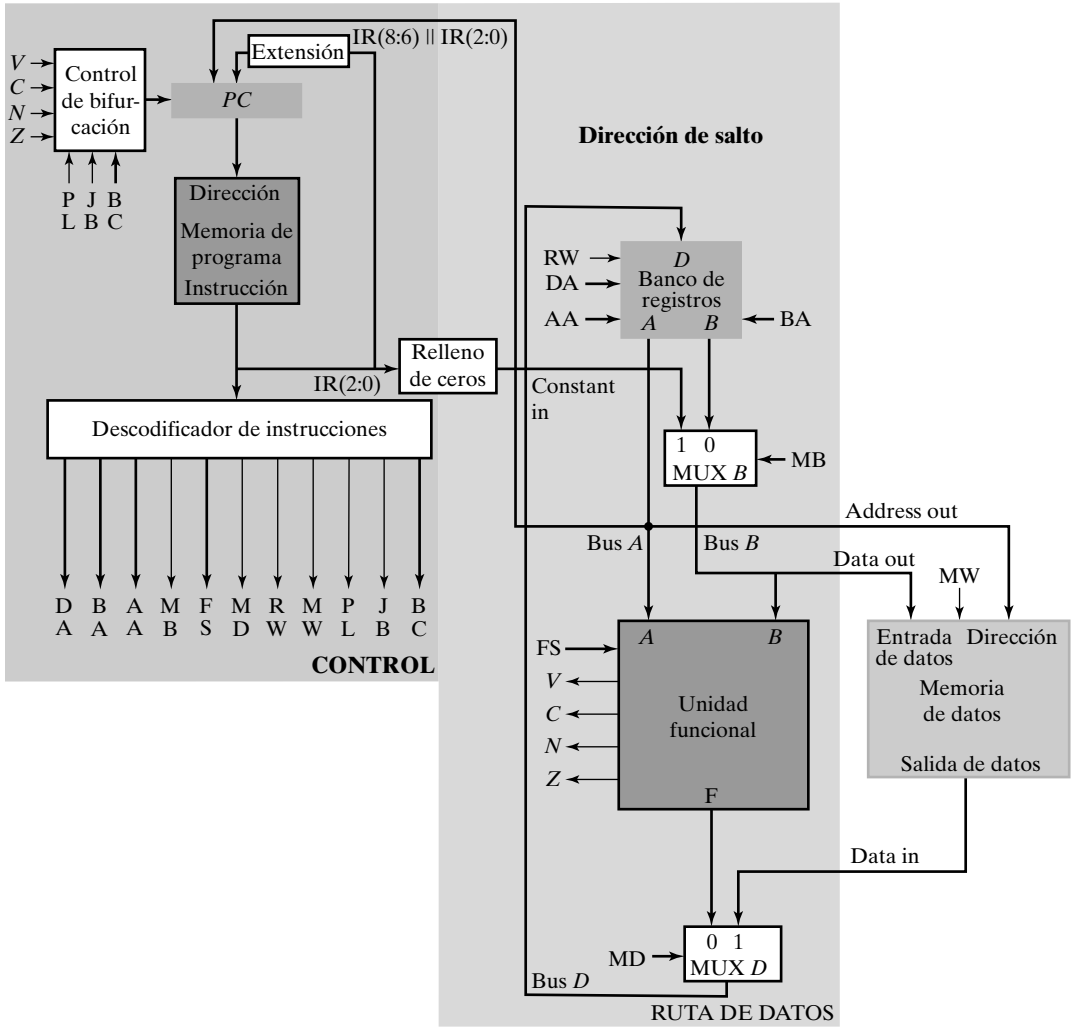
La ubicación de las instrucciones en la memoria, según se muestra en la Tabla 10-9 es bastante arbitraria. En muchos procesadores, la longitud de la palabra está entre 32 y 64 bits, de forma que las instrucciones pueden contener operandos inmediatos y direcciones mucho más largas que las propuestas aquí. Dependiendo de la arquitectura del procesador, algunos formatos de las instrucciones pueden ocupar dos o más palabras de memoria consecutivas. Además, el número de registros es, con frecuencia, mayor, de forma que los campos de la instrucción destinados a los registros deben contener más bits.

Llegados a este punto, es muy importante reconocer las diferencias entre la operación de un procesador y la microoperación hardware. Una operación está especificada por una instrucción que está almacenada en binario en la memoria del procesador. La unidad de control del procesador usa la dirección o direcciones proporcionadas por el contador de programa para recuperar la instrucción de la memoria. Luego se descodifican los bits del código de operación y otro tipo de información en la instrucción para realizar las microoperaciones necesarias para la ejecución de la instrucción. Por el contrario, una microoperación se especifica mediante los bits de una palabra de control del hardware, que se decodifica por el hardware del procesador para ejecutar la microoperación. La ejecución de una operación en el procesador suele necesitar una secuencia o programa de microoperaciones, en lugar de una única microoperación.

10-8 CONTROL CABLEADO DE UN SOLO CICLO

En la Figura 10-15 se muestra el diagrama de bloques de un procesador que tiene una unidad de control cableada y que trae y ejecuta una instrucción en un solo ciclo de reloj. A este procesador le llamaremos procesador de un solo ciclo. En la sección anterior se han presentado los recursos de almacenamiento, los formatos de las instrucciones y las especificaciones de las instrucciones. La ruta de datos mostrada es la misma que aparece en la Figura 10-11, con $m = 3$ y $n = 16$. La memoria de datos *M* está conectada a la ruta de datos mediante los buses Address Out, Data out y Data in. Tiene una señal de control, MW, que se pone a 1 para escribir en la memoria y a 0 en el caso contrario.

La unidad de control aparece a la izquierda de la Figura 10-15. Aunque no es habitual que la memoria de instrucciones sea parte de la unidad de control, junto con sus entradas de direcciones y salidas de instrucciones, por conveniencia se muestra junto con la unidad de control. En teoría, no vamos a escribir en la memoria de instrucciones, que funcionará como un circuito combinacional en lugar de funcionar como un componente secuencial. Como se estudió anteriormente, el *PC* proporciona la dirección de las instrucciones para las instrucciones que hacen uso de la memoria, y la salida de las instrucciones de la memoria de instrucciones va a la lógica de control, que en este caso es un decodificador de instrucciones. Las salidas de la memoria de instrucciones también va a parar el bloque de Extensión y Relleno de Ceros, que proporcionan la dirección relativa al *PC* y a la entrada de constantes, Constant in, de la ruta de datos respectivamente. El bloque Extensión añade a continuación del bit más a la izquierda del campo AD de 6 bits de dirección relativa hasta la izquierda de AD, preservando su representación en complemento a dos. El bloque de Relleno de Ceros añade 13 ceros a la izquierda del campo del



□ **FIGURA 10-15**
Diagrama de bloques de un procesador de un solo ciclo

operando (OP) de la instrucción para formar un operando de 16 bits sin signo, para utilizarlo en la ruta de datos. Por ejemplo, el valor del operando 110 pasa a ser 0000000000000110 o +6.

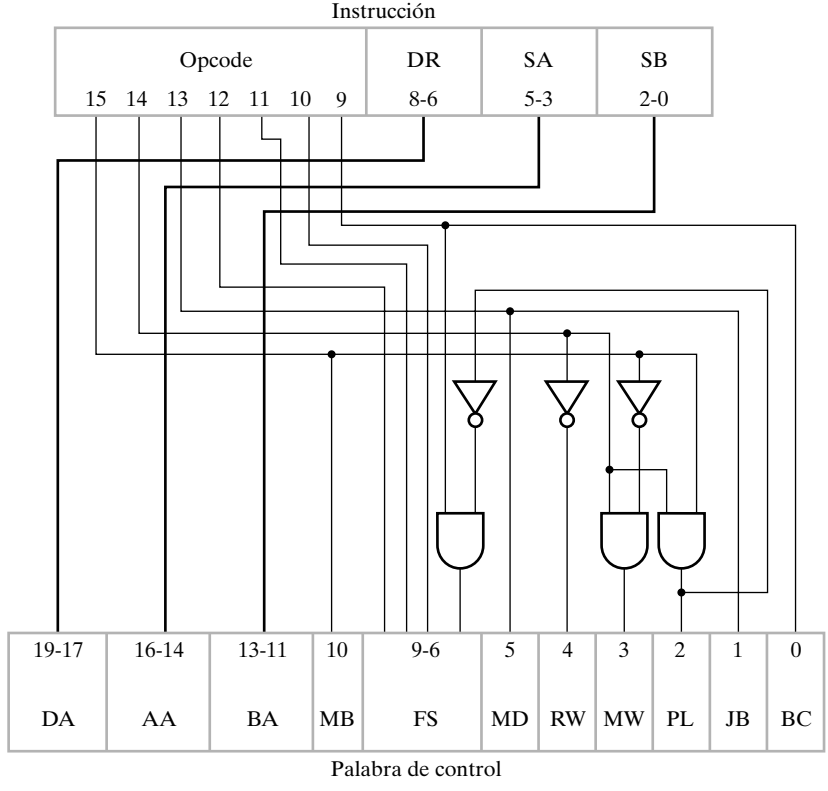
El contenido del *PC* se actualiza en cada ciclo de reloj. El comportamiento del *PC*, que es un registro complejo, se determina por el código de operación, *N* y *Z*, ya que *C* y *V* no se usan en el diseño de la unidad de control. Si tiene lugar un salto, el nuevo contenido del *PC* pasa a ser el del valor del Bus *A*. Si se toma una bifurcación, el nuevo valor del *PC* es la suma de valor previo del *PC* y la extensión de signo de la dirección relativa que, al estar representada en complemento a dos, puede ser positivo o negativo. En el caso contrario, el *PC* se incrementa en 1. Un salto se realiza si el bit 13 de la instrucción es igual a 1. Si el bit 13 es igual a 0 se efectúa una bifurcación condicional. El bit de status que afecta a la condición se selecciona con el bit 9 de la instrucción. Si el bit 9 es igual a 1, se selecciona el bit de status *N*, si es 0, se selecciona el *Z*.

Todas las partes del procesador que son secuenciales se muestran en gris. Véase que no hay lógica secuencial en la parte de control aparte del *PC*. Así, aparte de la proporcionar las direc-

ciones a la memoria de instrucciones, la lógica de control es en este caso combinacional. De hecho, combinada con la estructura de la ruta de datos y el uso de memorias separadas para instrucciones y datos, permite al procesador de un solo ciclo obtener y ejecutar una instrucción de la memoria de programa, todo en un solo ciclo de reloj.

Descodificador de instrucciones

El decodificador de instrucciones es un circuito combinacional que proporciona todas las palabras de control de la ruta de datos, basadas en el contenido de los campos de la instrucción. Un número de campos de la palabra de control se puede obtener directamente de los contenidos de los campos de la instrucción. Observando la Figura 10-16, vemos que los campos de control DA, AA y BA son iguales a los de los campos de la instrucción DR, SA y SB, respectivamente. Además, el campo de control BC, para la selección del bit de status para la condición de bifurcación, se toma directamente del último bit del código de operaciones. El resto de los campos de la palabra de control incluyen los bits de control de la ruta de datos y la memoria de datos, MB, MD, RW y MW. Además, hay dos bits para el control del *PC*, PL y JB. Si va a suceder un salto o una bifurcación, PL = 1, cargando el PC. Para PL = 0, el *PC* se incrementa. Con PL = 1, si JB = 1 avisa de la ocurrencia de un salto o bifurcación, y si JB = 0 anuncia una bifurcación condicional. Algunos de los campos de la palabra de control de un solo bit necesitan lógica para su realización. En orden a diseñar esta lógica, dividimos las diversas instrucciones



□ **FIGURA 10-16**
Diagrama del decodificador de instrucciones

posibles de un procesador sencillo en diferentes tipos de función y, posteriormente, se asignan los primeros tres bits del código de operación en los diversos tipos. Estos tipos de instrucciones según su función se muestran en la Tabla 10-10 se basan en el uso concreto de los recursos hardware del procesador, tales como el MUX *B*, la Unidad Funcional, el Banco de registros, la Memoria de Datos y el *PC*. Por ejemplo, el primer tipo de funciones utilizan la ALU, cambia MUX *B* para usar el Banco de Registros fuente, cambia MUX *D* para utilizar la salida de la unidad Funcional y escribir en el Banco de Registros. Otros tipos de instrucción según su función se definen como diversas combinaciones del uso de una entrada constante en lugar de un registro, lecturas y escrituras de la memoria de datos, y la manipulación del *PC* para saltos y bifurcaciones.

Viendo la relación entre los tipos de instrucción según su función, y los valores de la palabra de control necesarios para su ejecución, los bits desde 15 al 13 y el bit 9 se asignaron según se muestra en la Tabla 10-10. Esta asignación intenta minimizar la lógica necesaria para diseñar el decodificador. Para llevar a cabo su diseño, los valores para todos los campos de un solo bit en la palabra de control se determinaron a partir de los tipos de función y presentados en la Tabla 10-10. Véase que hay varias entradas indiferentes, X. Tratando la Tabla 10-10 como una tabla de verdad y optimizando las funciones lógicas, resulta la lógica para las salidas de un solo bit del decodificador de instrucciones de la Figura 10-16. En la optimización, los cuatro códigos no utilizados para los bits 15, 14, 13 y 9 se supusieron que tenían valores X para todos los campos de un solo bit. Esto implica que si uno de estos códigos aparece en un programa, su efecto será desconocido. Un diseño más conservador especifica RW, MW y PL todos a cero para estos cuatro códigos para asegurar que el estado del recurso de almacenamiento permanece sin cambiar para estos códigos no utilizados. La lógica resultante de esta optimización se muestra en la Figura 10-16 para realizar MB, MD, RW, MW, PL y JB.

□ **TABLA 10-10**
Tabla de verdad para la lógica de decodificador de instrucciones

	Bits de la instrucción				**Bits de la palabra de control**						
Instrucción según su función	**15**	**14**	**13**	**9**	**MB**	**MD**	**RW**	**MW**	**PL**	**JB**	**BC**
Operaciones de la unidad funcional usando registros	0	0	0	X	0	0	1	0	0	X	X
Lectura de memoria	0	0	1	X	0	1	1	0	0	X	X
Escritura de memoria	0	1	0	X	0	X	0	1	0	X	X
Operaciones de la unidad funcional usando registro y constante	1	0	0	X	1	0	1	0	0	X	X
Bifurcación condicional en cero (*Z*)	1	1	0	0	X	X	0	0	1	0	0
Bifurcación condicional en negativo (*N*)	1	1	0	1	X	X	0	0	1	0	1
Salto incondicional	1	1	1	X	X	X	0	0	1	1	X

La lógica restante del decodificador se reparte con el campo FS. Para todas las instrucciones, excepto las instrucciones de salto incondicional y de bifurcación condicional, los bits desde el 9 al 12 se ponen directamente para formar el campo FS. Durante las operaciones de bifurcación condicional, como la de bifurcación sobre cero, el valor del registro fuente *A* se debe pasar a través de la ALU de forma que los bits de status *N* y *Z* se puedan evaluar. Para esto hace falta que FS = 0000. Sin embargo, el uso del bit 9 para la selección de las bifurcaciones condicionales, necesita a veces que dicho bit, que controla el bit más a la derecha de FS sea 1. La contradicción entre los valores del bit 9 y FS se resuelve añadiendo una habilitación al bit 9 que fuerza FS_0 a cero siempre y cuando PL = 1, según se muestra en la Figura 10-16.

Ejemplo de instrucciones y programa

En la Tabla 10-11 se enumeran seis instrucciones para un procesador de un solo ciclo. Los nombres simbólicos asociados con las instrucciones son útiles para listar programas en forma simbólica mejor que en código binario. Debido a la importancia de la decodificación de las instrucciones, las seis columnas más a la derecha de la tabla muestran los valores de las señales críticas para cada instrucción, basándose en los valores obtenidos, utilizando la lógica de la Figura 10-16. Ahora supongamos que la primera, «Suma Inmediata» (ADI), se coloca en la salida de la memoria de instrucciones mostrada en la Figura 10-15. Luego, basándose en los tres primeros bits del código de operación, 100, las salidas del decodificador de instrucciones tendrán los valores MB = 1, MD = 0, RW = 1 y MW = 0. Los últimos tres bits de la instrucción $OP_{2\text{-}0}$, se extienden a 16 bits añadiendo ceros. Designaremos a esto mediante una sentencia de transferencia de registro con zf (del inglés *zero fill*). Como MB es 1, este valor rellenado con ceros se coloca en el Bus *B*. Con MD igual a 0, se selecciona la salida de la unidad funcional y, como los últimos cuatro bits del código de operación, 0010, especifican el campo FS, la operación es $A + B$. Así que el valor rellenado con ceros en el Bus *B* se suma al contenido del registro SA, presentando el resultado en el Bus *D*. Como RW = 1, el valor en el Bus *D* se escribe en el registro DR. Finalmente, como MW = 0, no se efectúa escritura en la memoria. La operación completa tiene lugar en un solo ciclo de reloj. Al comienzo de siguiente ciclo, el registro destino se escribe y, PL = 0, el PC se incrementa para apuntar a la siguiente instrucción.

La segunda instrucción, LD, se carga de la memoria con el código de operación 0010000. Los primeros tres bits de este código, 001, dan los valores de control MD = 1, RW = 1 y MW = 0. Estos valores, más el campo del registro fuente SA y el registro de destino DR, totalmente especificado en esta instrucción, cargan el contenido de la dirección de memoria especificada por el registro SA en el registro DR. De nuevo, como PL = 0, el *PC* se incrementan. Véase que los valores JB y BC se ignoran ya que no es ni una instrucción de salto ni de bifurcación.

La tercera instrucción, ST, almacena el contenido de un registro en la memoria. Los tres bits primeros del código de operación, 010, dan a las siguientes señales de control los valores MB = 0, RW = 0 y MW = 1. Cuando MW = 1 se produce una operación de escritura en la memoria, donde la dirección y el dato proceden del banco de registros. Con RW = 0 se evita que se escriba en el banco de registros. Las direcciones de escritura en la memoria proceden del registro seleccionado en el campo SA, y el dato a escribir en la memoria procede del registro seleccionado en SB, ya que MB = 0. El campo DR, aunque está presente, no se usa al no producirse escritura a un registro.

Como este procesador tiene instrucciones de carga y almacenamiento, y no cambian la carga y almacenamiento de operandos con otras operaciones, se dice que tiene una arquitectura de carga/almacenamiento. El uso de tal arquitectura simplifica la ejecución de las instrucciones.

□ **TABLA 10-11**
Seis instrucciones para el procesador de un solo ciclo

Código de operación	Nombre simbólico	Formato	Descripción	Función	MB	MD	RW	MW	PL	JB	BC
1000010	ADI	Inmediato	Suma inmediata de un operando	R[DR] ← R[SA] + zf I(2:0)	1	0	1	0	0	0	0
0010000	LD	Registro	Carga el contenido de memoria en registro	R[DR] ← M[R[SA]]	0	1	1	0	0	1	0
0100000	ST	Registro	Almacena el contenido de un registro en memoria	M[R[SA]] ← R[SB]	0	1	0	1	0	0	0
0001110	SL	Registro	Desplaza a la izquierda	R[DR] ← sl R[SB]	0	0	1	0	0	1	0
0001011	NOT	Registro	Complement register	R[DR] ← $\overline{R[SA]}$	0	0	1	0	0	0	1
1100000	BRZ	Salto/bifurcación	Si R[SA] = 0, bifurcación a PC + se AD	Si R[SA] = 0, PC ← PC + se AD, Si R[SA] ≠ 0, PC ← PC + 1	1	0	0	0	1	0	0

Las dos siguientes instrucciones utilizan la unidad funcional y escriben en el banco de registros sin operandos inmediatos. Los últimos cuatro bits del código de operación, el valor para el campo FS de la palabra de control, especifican una operación de la unidad funcional. En estas dos instrucciones involucra a un único registro fuente, R[SA] para la operación NOT y R[SB] para el desplazamiento a la izquierda, y un registro destino.

La última instrucción es una bifurcación condicional y manipula el valor de *PC*. Tiene PL = 1, haciendo que se cargue el contador de programa en lugar de incrementarse, y JB = 0, haciendo una bifurcación condicional en lugar de un salto. Al ser BC = 0, se verifica si el registro R[SA] es cero. Si R[SA] es igual a cero, el contenido *PC* pasa a ser *PC* + se AD, que indica una extensión de signo. En otro caso, el *PC* se incrementa. En esta instrucción, los campos DR y SB pasan a ser el campo de direcciones de 6 bits AD, que utiliza extensión de signo y se suma al contenido de *PC*.

Para demostrar cómo instrucciones tales como éstas se pueden usar en un sencillo programa, considere la expresión aritmética 83 − (2 + 3). El siguiente programa realiza este cálculo, suponiendo que el registro *R*3 tiene el valor 248, la posición 248 de la memoria de datos contiene un 2, la posición 249 guarda el valor 83 y el resultado se guarda en la posición 250.

LD	*R*1, *R*3	Carga a *R*1 con el contenido de la posición 248 de la memoria (*R*1 = 2)
ADI	*R*1, *R*1, 3	Suma 3 a *R*1 (*R*1 = 5)
NOT	*R*1, *R*1	Complementa a *R*1
INC	*R*1, *R*1	Incrementa a *R*1 (*R*1 = −5)
INC	*R*3, *R*3	Incrementa el contenido de *R*3 (*R*3 = 249)
LD	*R*2, *R*3	Carga a *R*2 con el contenido de la posición 249 de la memoria (*R*2 = 83)
ADD	*R*2, *R*2, *R*1	Suma el contenido de *R*1 al contenido de *R*2 (*R*2 = 78)
INC	*R*3, *R*3	Incrementa el contenido de *R*3 (*R*3 = 250)
ST	*R*3, *R*2	Almacena el contenido de *R*2 en la posición 250 de memoria (M[250] = 78)

En este caso, la substracción se hace tomando el complemento a 2 de (2 + 3) y sumándolo a 83; se podría haber utilizado también la operación de substracción, SUB. Si el campo de un registro no se utiliza en la ejecución de una instrucción, se omite su valor simbólico. Los valores simbólicos para una instrucción de tipo registro, si es que hay alguno, se colocan en el orden siguiente DR, SA y SB. En las instrucciones de tipo inmediato, los campos están en el orden DR, SA y OP. Para almacenar este programa en la memoria de programa es necesario convertir todo los nombres simbólicos y los números decimales utilizados en sus códigos binarios correspondientes.

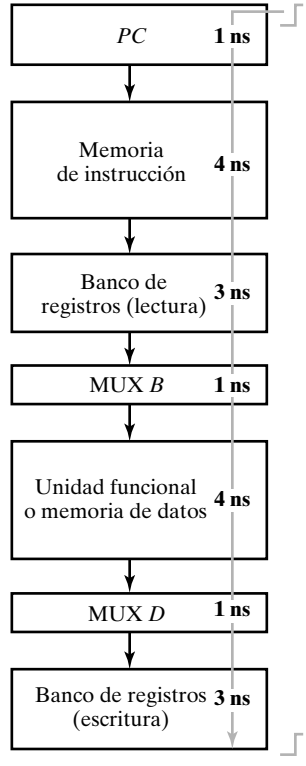
Problemas del procesador de un solo ciclo

Aunque puede haber casos en los que la estrategia de temporización y control del procesador de un solo ciclo es útil, presenta ciertas limitaciones. Una limitación es a la hora de realizar operaciones complejas. Por ejemplo, supongamos que desea tener una instrucción que ejecute la multiplicación binaria usando un algoritmo de suma y desplazamiento. Con la ruta de datos dada, esta operación no se puede realizar mediante una microoperación que se pueda ejecutar en un

solo ciclo de reloj. Pare ello se necesita una organización del control que proporcione varios ciclos de reloj para la ejecución de las instrucciones.

Además, el procesador de un solo ciclo de reloj tiene dos memorias de 16 bits, una para las instrucciones y otra para los datos. En un procesador sencillo, con las instrucciones y los datos en la misma memoria de 16 bits, para ejecutar una instrucción que carga un dato de la memoria a un registro, se necesita realizar dos accesos de lectura. El primer acceso obtiene la instrucción y el segundo, caso de que sea necesario, lee o escribe el dato. Puesto que se deben aplicar dos direcciones diferentes a las entradas de las memorias, se necesitan al menos dos ciclos de reloj para obtener y ejecutar la instrucción. Esto se puede lograr fácilmente con el control de varios ciclos de reloj.

Para terminar, el procesador de un solo ciclo de reloj tiene un límite más bajo de periodo de reloj debido al camino con el retardo más grande. Este camino se muestra en azul en el diagrama simplificado de la Figura 10-17. El retardo total a lo largo del camino es de 17 ns. Esto limita la frecuencia de reloj a 58.8 MHz, que, aunque pueda ser adecuada para ciertas aplicaciones, es demasiado lenta para una CPU de un procesador moderno. Para conseguir frecuencias de reloj más altas, se debe reducir el retardo de los componentes que forman el camino o el número de estos. Si los retardos de los componentes no se pueden reducir, la única alternativa es reducir el número de componentes de la ruta. En el Capítulo 12, la técnica de *pipelining* permite reducir el número de componentes en la ruta combinacional con el retardo más largo, permitiendo incrementar la frecuencia de reloj. En el Capítulo 12 se presenta una ruta de datos y su control en *pipeline*, demostrándose el rendimiento que se puede obtener de la CPU.

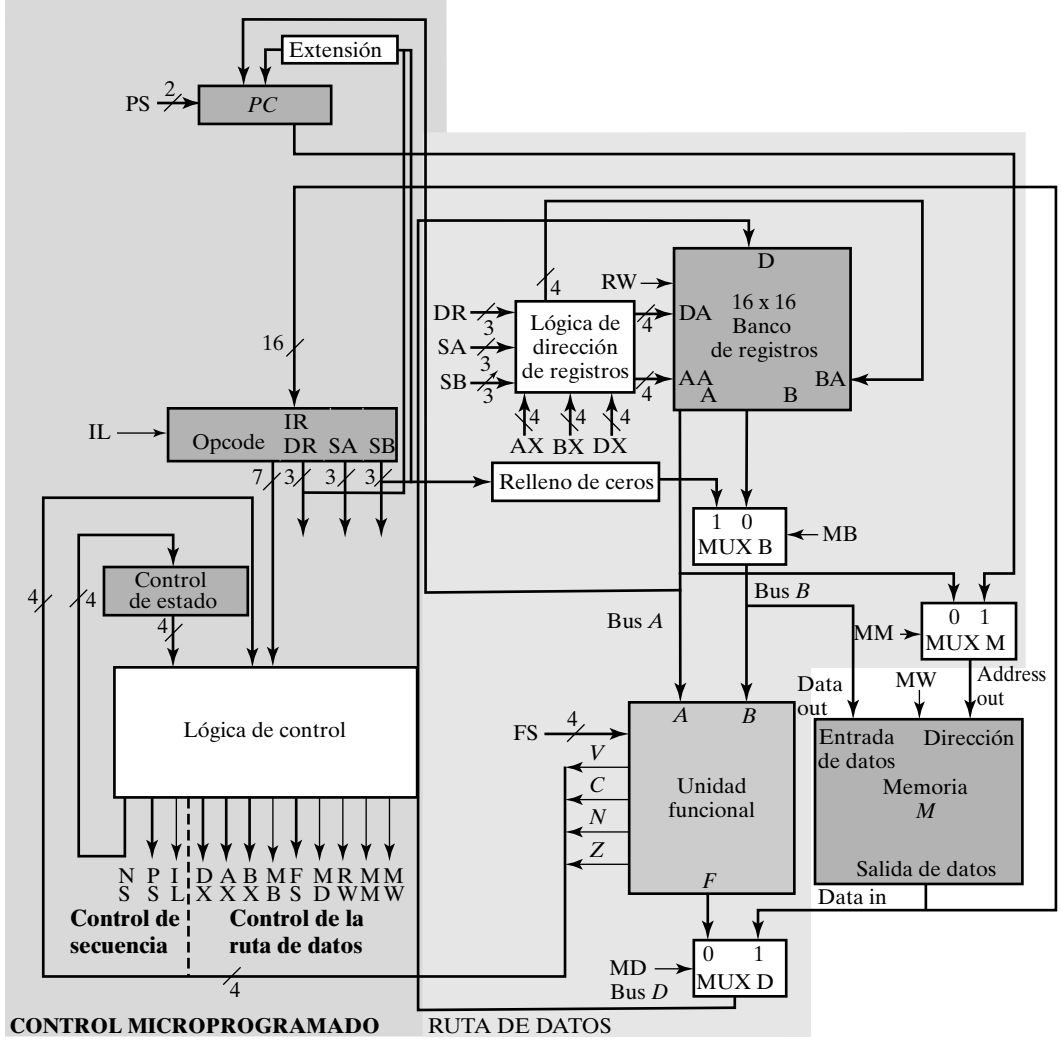


□ **FIGURA 10-17**
Retardo más largo en un procesador de un solo ciclo

10-9 CONTROL CABLEADO MULTICICLO

Para ver el control en varios ciclos vamos a usar la arquitectura de un sencillo procesador pero con su ruta de datos, memoria y control modificados. El objetivo de las modificaciones es mostrar el uso de una sola memoria, tanto para datos como para instrucciones, y para ver cuan complejas son las instrucciones que se pueden realizar utilizando varios ciclos de reloj por instrucción. El diagrama de bloques de la Figura 10-18 muestra las modificaciones en la ruta de datos, la memoria y el control.

Los cambios en el procesador de un solo ciclo se pueden observar comparando las Figuras 10-15 y 10-18. La primera modificación, posible pero no indispensable, con operaciones en varios ciclos de reloj es reemplazar la memoria de programa y de datos de la Figura 10-15, que está separada, por una única memoria, *M*, como se muestra en la Figura 10-18. Para traer las



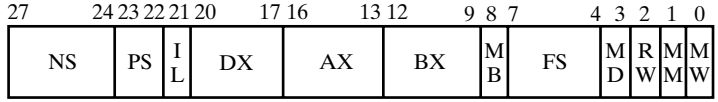
□ **FIGURA 10-18**
Diagrama de bloques de un procesador de varios ciclos de reloj

instrucciones, el *PC* tiene la dirección fuente para la memoria y, para acceder a los datos es el Bus *A* el que tiene la dirección fuente. A la entrada de direcciones de la memoria, el multiplexor MUX *M* selecciona entre estas dos fuentes de direcciones. El MUX *M* necesita una señal de control adicional, MM, que se añade al formato de la palabra de control. Puesto que la unidad de control necesita las instrucciones de la memoria *M*, se ha añadido una ruta desde sus salidas hasta el registro de instrucciones, IR, de la unidad de control.

En la ejecución de una instrucción mediante varios ciclos de reloj, los datos generados en el ciclo en curso se necesitan generalmente en el ciclo posterior. Estos datos puedes almacenarse temporalmente en un registro en el momento en que se generan y mantenerlos hasta el momento en que se utilizan. Los registros utilizados en dicho almacenamiento temporal son, generalmente, no visibles para el usuario (es decir, no forman parte de los recursos de almacenamiento). La segunda modificación proporciona estos registros de almacenamiento temporal duplicando el número de registros en el banco de registros. Los registros de 0 hasta 7 forman parte de los recursos de almacenamiento y los registros de 8 hasta 15 son sólo para almacenamiento temporal durante la ejecución de las instrucciones, por tanto, no forman parte de los recursos de almacenamiento visibles al usuario. El direccionamiento de 16 registros necesita 4 bits y pasa a ser más complejo ya que el direccionamiento de los ocho primeros registros se deben controlar a partir de la instrucción y la unidad de control, y los ocho registros siguientes se controlan desde la unidad de control. Esto se maneja mediante el registro de direcciones lógicas de la Figura 10-18 y modificando los campos de la palabra de control DX, AX y BX. Los detalles de este cambio se discutirán posteriormente cuando se haya definido el control.

El *PC* es el único componente de la unidad de control que se mantiene aunque también debe ser modificado. Durante la ejecución de una instrucción multiciclo, el *PC* debe mantenerse en su valor actual durante todos los ciclos excepto uno. Para proporcionar esta capacidad, así como una operación de incremento y dos operaciones de carga, el *PC* se modifica para controlarse mediante un campo de la palabra de control de 2 bits, *PS*. Como el *PC* se controla completamente mediante la palabra de control, la lógica de control de salto condicional, representada anteriormente mediante BC, se absorbe por el bloque Control Logico de la Figura 10-18. Debido a que el procesador modificado es multiciclo, la instrucción necesita ser guardada en un registro, para utilizarse durante su ejecución, ya que su valor se necesitará probablemente en más de un ciclo y no sólo en el primero. El registro utilizado para este propósito es el *registro de instrucciones IR* de la Figura 10-18. Ya que el *IR* se carga solamente cuando una instrucción se empieza a leer de la memoria, tiene una señal de Load Enable, IL, que se añade a la palabra de control. Debido a que la operación necesita varios ciclos, es preciso tener un circuito de control secuencial, que pueda proporcionar una secuencia de palabras de control para las microoperaciones, para interpretar las instrucciones y reemplazar al decodificador de instrucciones. La unidad secuencial de control está formada por el registro de control de estado y la lógica de control combinacional. La lógica de control tiene el estado, el código de operación, y los bits de status como entradas y produce la palabra de control como salida. Conceptualmente, la palabra de control se divide en dos partes, una para la secuencia de control, que es el siguiente estado de toda la unidad de control, y otra para el control de la ruta de datos, que controla las microoperaciones ejecutadas mediante la ruta de datos y la Memoria *M*, según se muestra en la Figura 10-18.

La palabra de control modificada de 28 bits se da en la Figura 10-19 y las definiciones de los campos y palabras de control de la palabra de control se dan en las Tablas 10-12 y 10-13. En la Tabla 10-12, los campos DX, AX y BX controlan la selección de los registros. Si el MSB de uno de estos campos es 0, entonces el registro correspondiente de direcciones DA, AA o BA es aquel dado mediante 0 || DR, 0 || SA, y 0 || SB respectivamente. Si el MSB de alguno de estos



□ **FIGURA 10-19**
Formato de la palabra de control de un procesador multiciclo

campos es 1, entonces la dirección del registro correspondiente es el contenido de los campos DX, AX y BX. Este proceso de selección se lleva a cabo mediante la lógica del registro de direcciones, que contiene tres multiplexores, uno por cada DA, AA y BA, controlado por el MSB de DX, AX y BX respectivamente. En la Tabla 10-12 también se dan los valores del código para el campo MM, que determina si Address out o PC sirve como dirección para la memoria *M*. Los campos restantes de la Tabla 10-12: MB, MD, RW y MW, tienen las mismas funciones que para los procesadores de un solo ciclo de reloj.

En el circuito secuencial de control, el registro de control de estado tiene un conjunto de estados, como tiene un conjunto de flip-flops de cualquier otro circuito secuencial. En este nivel de discusión, suponemos que cada estado tiene un nombre abstracto que se puede utilizar como valor del estado actual y del estado futuro. En el proceso de diseño, se necesita realizar una asignación de estados a estos estados abstractos. Según la Tabla 10-13, el campo NS de la palabra de control indica el siguiente estado del registro de control de estado. Hemos asignado cuatro bits para el código del estado, aunque esto se puede modificar según sea necesario, dependiendo del número de estados necesarios y la sentencia de asignación utilizada en el diseño. Este campo en particular, podría considerarse como esencial para el control y el circuito secuencial y como parte de la palabra de control, aunque aparecerá en la tabla de estados del control en cualquier caso. El campo de 2 bits, PS, controla el contador de programa, *PC*. Para un ciclo de reloj determinado, el *PC* contiene su estado (00), incrementa su estado en 1 (01), carga condicionalmente el *PC* con extensión de signo AD (10), o carga incondicionalmente el contenido de *R*[SA] (11). Por último, se carga el registro de instrucciones sólo una vez durante la ejecución de una instrucción. Así, en cualquier ciclo, o se carga una nueva instrucción (IL = 1) o la instrucción permanece sin cambiar (IL = 0).

Diseño del control secuencial

El diseño de un circuito de control secuencial se puede realizan usando las técnicas presentadas en los Capítulo 6 y 8. Sin embargo, comparado con los ejemplos de esta parte, incluso para este procesador relativamente sencillo, el control es bastante complicado. Suponiendo que hay cuatro variables, la lógica combinacional del control tiene 15 variables de entrada y 28 variables de salida. Esto indica que una tabla de estados condensada no es demasiado difícil de hacer, pero un diseño manual detallado de la lógica es muy complejo, haciendo que el uso de una PLA o de síntesis lógica sean unas de las opciones más viables. Por tanto, como consecuencia de esto, nos centraremos en el desarrollo de tablas de estados en lugar de diseñar una lógica detallada.

Empezamos desarrollando el diagrama ASM que representa las instrucciones que se pueden realizar con el mínimo número de ciclos de reloj. Las extensiones del diagrama pueden desarrollarse para la realización de instrucciones que requieren más que el número mínimo de ciclos de reloj. Los diagramas ASM proporcionan la información detallada para realizar las entradas de la tabla de estados para diseñar el conjunto de instrucciones. Las instrucciones que necesitan un acceso a memoria para los datos y también para las mismas instrucciones requieren al menos

□ **TABLA 10-12**
Información de la palabra de control para la ruta de datos

DX	AX	BX	Código	MB	Código	FS	Código	MD	RW	MM	MW	Código
R[DR]	*R*[SA]	*R*[SB]	0XXX	Registro	0	$F = A$	0000	Unidad funcional	No escribe	Dirección de salida	No escribe	0
*R*8	*R*8	*R*8	1000	Constante	1	$F = A + 1$	0001	Data In	Escribe	PC	Escribe	1
*R*9	*R*9	*R*9	1001			$F = A + B$	0010					
*R*10	*R*10	*R*10	1010			Sin usar	0011					
*R*11	*R*11	*R*11	1011			Sin usar	0100					
*R*12	*R*12	*R*12	1100			$F = A + \bar{B} + 1$	0101					
*R*13	*R*13	*R*13	1101			$F = A - 1$	0110					
*R*14	*R*14	*R*14	1110			Sin usar	0111					
*R*15	*R*15	*R*15	1111			$F = A \wedge B$	1000					
						$F = A \vee B$	1001					
						$F = A \oplus B$	1010					
						$F = \bar{A}$	1011					
						$F = B$	1100					
						$F = \text{sr } B$	1101					
						$F = \text{sl } B$	1110					
						Sin usar	1111					

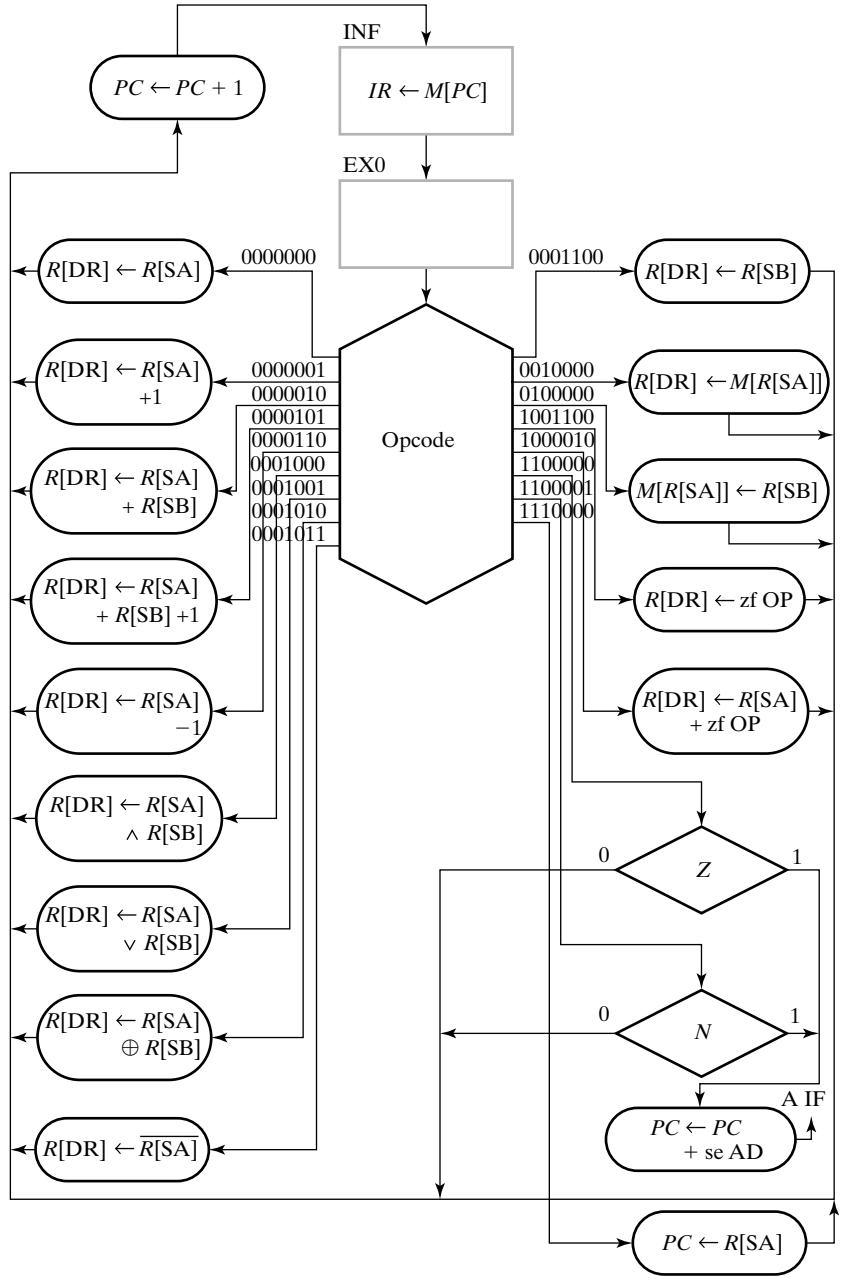
□ **TABLA 10-13**
Control de la información para la secuencia de control

NS	PS		IL	
Siguiente estado	**Acción**	**Código**	**Acción**	**Código**
Da el siguiente estado del registro de control de estado	Mantiene el PC	00	No carga	0
	Incrementa el PC	01	Carga instrucción	1
	Bifurcación	10		
	Salto	11		

dos ciclos de reloj. Es conveniente separar los ciclos de reloj en un procedimiento en dos pasos: traer la instrucción y ejecutar la instrucción. Según esta división, en la Figura 10-20 se da el diagrama ASM para instrucciones de dos ciclos. La lectura de la instrucción ocurre en el estado INF, arriba del diagrama. El *PC* contiene la dirección de la instrucción en memoria *M*. Esta dirección se aplica a la memoria y la palabra que se lee de la memoria se carga en *IR* en el pulso de reloj que termina el estado INF. El mismo pulso de reloj provoca que el nuevo estado pase a ser EX0. En el estado EX0, la instrucción se descodifica mediante el uso de una gran caja de decisión vectorial y las microoperaciones ejecutando todo o parte de la instrucción que aparece en la caja de salida condicional. Si la instrucción se puede completar en el estado EX0, el siguiente estado en preparar la lectura de la siguiente instrucción es INF. Además, para las instrucciones que no cambian el contenido del *PC* durante su ejecución, el *PC* se incrementa. Si se necesitan estados adicionales para la ejecución de las instrucciones, el siguiente estado es EX1. En cada uno de los estados de ejecución hay 128 combinaciones posibles diferentes, basadas en el código de operación. Cuando se utilizan los bits de status, típicamente uno solo cada vez, la salida de la caja de decisión vectorial alimenta a una o más cajas de decisión escalares, según se ilustra en la instrucción de salto condicional en la parte baja derecha de la Figura 10-20.

A continuación describimos un ejemplo de ejecución de una instrucción especificada por el diagrama ASM de la Figura 10-20. El primer código de operaciones es 000000 para la instrucción «mover A», (MOVA). Esta instrucción involucra una sencilla transferencia desde el registro fuente A al registro de destino, como se especifica mediante la transferencia de registro mostrada en el estado EX0 para el código de la instrucción. Aunque los bits de estado *N* y *Z* son válidos, no se usan en la ejecución de esta instrucción. El *PC* se incrementa en el flanco de reloj al terminar el estado EX0, una acción que ocurre para todas las instrucciones excepto para las instrucciones de saltos incondicionales y saltos condicionales del diagrama ASM.

El tercer código de operación 0000010 es la instrucción ADD con la transferencia de registro para la suma mostrada. En este caso, los bits de status *V, C, N* y *Z* son válidos, aunque no se usan. El código undécimo, 0010000, es la instrucción de carga (LD), que utiliza el valor del registro especificado por SA para las direcciones y las cargas de datos desde la memoria *M* al registro especificado por DR. El duodécimo código de operación, 010000, es para la instrucción de almacenamiento (ST), que almacena el valor en el registro SB en la posición de memoria *M* especificada por la dirección del registro SA. El décimo cuarto código de operación, 1001100, es la suma inmediata (ADI), que suma el valor rellenado con ceros del campo OP, los tres bits más a la derecha de la instrucción, al contenido del registro SA y coloca el resultado en el registro DR.



□ **FIGURA 10-20**
Diagrama básico ASM para procesadores multiciclo

El código de operación décimo sexto, 1100001, es la instrucción de salto condicional sobre negativo (BRN). La decodificación de esta instrucción hace que el valor del registro especificado por SA se pase a través de la unidad funcional para evaluar el valor de los bits de status *N* y *Z*. Los valores de *N* y *Z*, vuelven a entrar en la lógica de control. Según el valor de *N*, el salto condicional se realiza o no sumando la dirección extendida AD desde la instrucción al valor del

PC o incrementa el *PC*, respectivamente. Esto se representa mediante la caja de decisión para *N*, mostrada en la Figura 10-20.

A partir de este diagrama ASM, se puede extraer la tabla de estados del circuito de control secuencial, según se muestra en la Tabla 10-14. Los estados actuales se dan con nombres abtractos y los opcodes y bits de status sirven de entradas. En el caso de los bits de status, sólo se especifican aquellos utilizados en la instrucción. Usando combinaciones de bits y varios patrones de bits de status, es posible especificar las funciones de los bits de status. Véase que muchas casillas de la Tabla 10-14 contienen X, simbolizando «indiferente». Para estos casos, la entrada o recurso que no se utiliza en una determinada microoperación o bits concretos del código que son X no se utilizan para controlarla. Es un ejercicio conveniente determinar cómo se obtiene cada una de las entradas de la Tabla 10-14, basándose en las Tablas 10-12, 10-13 y la Figura 10-20.

Es interesante comparar brevemente la temporización de la ejecución de las instrucciones en esta organización con las del procesador de un solo ciclo. Cada instrucción necesita dos ciclos de reloj para acceder y ejecutarse, comparado con un ciclo de reloj que necesita el procesador de un solo ciclo. Debido al larguísimo retardo de la ruta desde el *PC* hasta la memoria de instrucciones, decodificador de instrucción, ruta de datos y control de bifurcación, terminado por el registro de instrucciones, los periodos de reloj son algo más cortos. Sin embargo, debido a los requisitos de tiempo de set-up de los flip-flops del *IR* y un posible desbalance en los retardos de algunas rutas a través del circuito, el tiempo total que se necesita para ejecutan una instrucción podría se tan largo o más que en un procesador de un solo ciclo. Entonces ¿cuál es la ventaja de esta organización que usa una sola memoria? Las dos siguientes instrucciones tienen la respuesta.

La primera instrucción a añadir es «carga un registro indirecta» (LRI), con opcode 0010001. En esta instrucción, el contenido del registro SA direcciona una palabra de la memoria. La palabra, que se conoce como una dirección indirecta, se usa para direccionar la palabra en la memoria que se cargó en el registro DR. Esto se puede representar simbólicamente como

$$R[\text{DR}] \leftarrow M[M[R[\text{SA}]]]$$

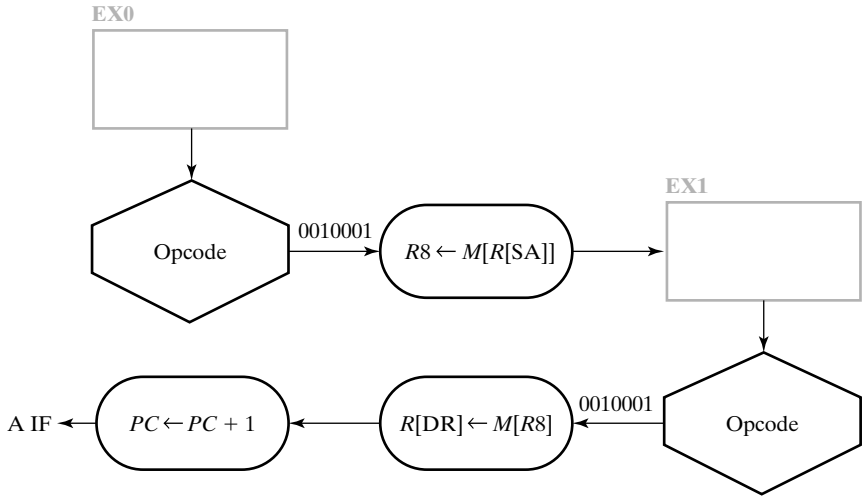
El diagrama ASM para la ejecución de esta instrucción se da en la Figura 10-21. Siguiendo al acceso de la instrucción, el estado pasa a ser EX0. En este estado, *R*[SA] direcciona la memoria para obtener la dirección indirecta, que se coloca después en el registro temporal *R*8. En el estado EX1, el siguiente acceso a memoria ocurre con la dirección en *R*8. El operando obtenido se coloca en *R*[DR] para completar la operación y el *PC* se incrementa. El ASM vuelve después al estado INF para acceder a la siguiente instrucción. La caja de decisión vectorial para el opcode se necesita en todos los estados, ya que estos mismos estados se usan en otras instrucciones para su ejecución. Claramente, con dos accesos a la memoria *M*, esta instrucción podría no ser ejecutada en un procesador de un solo ciclo o utilizar dos ciclos de reloj en un procesador de varios ciclos de reloj. Además, para evitar sobrescribir el contenido de los registros *R*0 a *R*7 (excepto para *R*[SA]), el uso de *R*8 como registro temporal es fundamental. La instrucción LRI proporciona una mejora en el tiempo de ejecución en el último caso.

Las últimas dos instrucciones a añadir son las de «desplazamiento múltiple a la derecha» (SRM) y «desplazamiento múltiple a la izquierda» (SLM), con opcode 0001101 y 0001110, respectivamente. Estas dos instrucciones pueden compartir la mayor parte de la secuencia de las microinstrucciónes que se utilizan. SRM especifica que el contenido del registro SA se tiene que desplazar a la derecha el número de posiciones dadas por los tres bits de campo OP, colocando el resultado en el registro DR. El diagrama ASM para esta operación (y para SLM) se

□ **TABLA 10-14**
Tabla de estados para instrucciones de dos ciclos

Estado	Entradas		Siguiente estado	Salidas											Comentarios
	Opcode	VCNZ		IL	PS	DX	AX	BX	MB	FS	MD	RW	MM	MW	
INF	XXXXXX	XXXX	EX0	1	00	XXXX	XXXX	XXXX	X	XXXX	X	0	1	0	$IR \leftarrow M[PC]$
EX0	0000000	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	0000	0	1	X	0	MOVA $R[DR] \leftarrow R[SA]$*
EX0	0000001	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	0001	0	1	X	0	INC $R[DR] \leftarrow R[SA] + 1$*
EX0	0000010	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	0010	0	1	X	0	ADD $R[DR] \leftarrow R[SA] + \overline{R[SB]}$*
EX0	0000101	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	0101	0	1	X	0	SUB $R[DR\} \leftarrow R[SA] + \overline{R[SB]} + 1$*
EX0	0000110	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	0110	0	1	X	0	DEC $R[DR] \leftarrow R[SA] + (-1)$*
EX0	0001000	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	1000	0	1	X	0	AND $R[DR] \leftarrow R[SA] \wedge R[SB]$*
EX0	0001001	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	1001	0	1	X	0	OR $R[DR] \leftarrow R[SA] \vee R[SB]$*
EX0	0001010	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	1010	0	1	X	0	XOR $R[DR] \leftarrow \overline{R[SA]} \oplus R[SB]$*
EX0	0001011	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	1011	0	1	X	0	NOT $R[DR] \leftarrow \overline{R[SA]}$*
EX0	0001100	XXXX	INF	0	01	0XXX	XXXX	0XXX	0	1100	0	1	X	0	MOVB $R[DR] \leftarrow R[SB]$*
EX0	0010000	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	XXXX	1	1	0	0	LD $R[DR] \leftarrow M[R[SA]]$*
EX0	0100000	XXXX	INF	0	01	XXXX	0XXX	0XXX	0	XXXX	X	0	0	1	ST $M[R[SA]] \leftarrow R[SB]$*
EX0	1001100	XXXX	INF	0	01	0XXX	XXXX	XXXX	1	1100	0	1	0	0	LDI $R[DR] \leftarrow$ zf OP*
EX0	1000010	XXXX	INF	0	01	0XXX	0XXX	XXXX	1	0010	0	1	0	0	ADI $R[DR] \leftarrow R[SA] +$ zf OP*
EX0	1100000	XXX1	INF	0	10	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRZ $PC \leftarrow PC +$ se AD
EX0	1100000	XXX0	INF	0	01	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRZ $PC \leftarrow PC + 1$
EX0	1100001	XX1X	INF	0	10	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRN $PC \leftarrow PC +$ se AD
EX0	1100001	XX0X	INF	0	01	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRN $PC \leftarrow PC + 1$
EX0	1110000	XXXX	INF	0	11	XXXX	0XXX	XXXX	X	0000	X	0	0	0	JMP $PC \leftarrow R[SA]$

* Para esta combinación de estado y entrada, también ocurre que $PC \leftarrow PC + 1$.



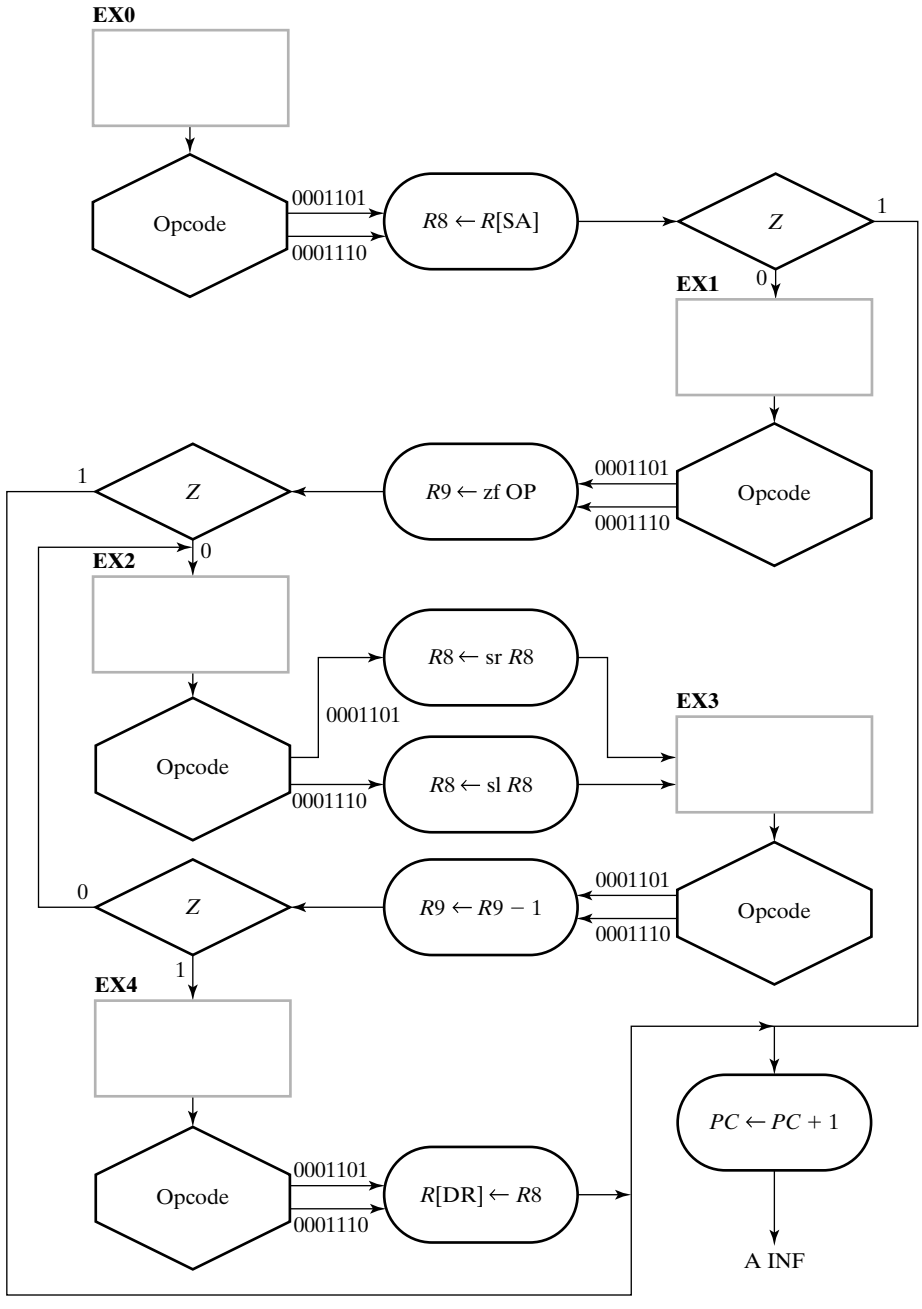
□ **FIGURA 10-21**
Diagrama ASM para la instrucción de registro indirecto

muestra en la Figura 10-22. El registro *R*9 almacena el número de bits restantes a desplazar, y el desplazamiento se realiza en el registro *R*8.

Inicialmente, el contenido de *R*[SA] a desplazar se coloca en *R*8. Según se carga en *R*8, se comprueba para ver si es 0 y, por tanto, no necesita ser desplazado. Así mismo, se comprueba si la cantidad a desplazar, cargada en el registro *R*9, es 0, indicando que no es necesario desplazar. Si se satisface cada caso, se completa la ejecución de instrucción, y el ASM vuelve al estado INF, en otro caso, se realiza una operación de desplazamiento a la derecha del contenido del registro *R*8. *R*9 se decrementa y se comprueba si es cero. Si $R9 \neq 0$, entonces se repiten el decremento y el desplazamiento. Si $R9 = 0$, entonces el contenido de *R*8 ya ha sido desplazado el número de posiciones especificadas por OP, y el resultado se transfiere a *R*[DR] completando la ejecución de la instrucción, y el ASM vuelve al estado INF.

Si tanto el operando como la cantidad a desplazar son distintas de cero, SRM, incluyendo su lectura, necesita $2s + 4$ ciclos de reloj, donde *s* es el número de posiciones desplazadas. El rango de ciclos de reloj necesario, incluyendo el acceso a la instrucción, va de 6 a 18. Si la misma operación fuese llevada a cabo mediante un programa utilizando la instrucción de desplazamiento, más el incremento y la bifurcación, entonces podrían hacer falta $3s + 3$ instrucciones y $6s + 6$ ciclos. La mejora en el número de ciclos de reloj necesarios es de $4s + 2$, así se ahorra entre 6 y 30 ciclos de reloj en un procesador multiciclo para un operando y número de desplazamientos distintos de cero. Además, se necesitan cinco posiciones menos de memoria para el almacenamiento de una instrucción SRM, en comparación con la del programa.

En el diagrama ASM de la Figura 10-22, los estados INF y EX0 (y EX1) son los mismos que los usados en las instrucciones de dos ciclos en el diagrama ASM de la Figura 10-20 para la instrucción LRI de la Figura 10-21. Además, la realización de la operación de desplazamiento a la izquierda se muestra en la Figura 10-22, en la que, basado en el opcode, el desplazamiento a la izquierda de *R*8 reemplaza al desplazamiento a la derecha de *R*8. Como consecuencia, la lógica que define los estados que realizan estas dos instrucciones se pueden compartir. Además, la lógica utilizada para la secuenciación de los estados se puede compartir entre las realizaciones de las instrucciones SRM y SLM.



□ **FIGURA 10-22**
Diagrama ASM para una instrucción de desplazamiento múltiple a la derecha

La especificación de la tabla de estados de la Tabla 10-15 se obtiene utilizando la información del diagrama ASM de la Figura 10-22 y las Tablas 10-12 y 10-13. Los códigos se extraen de la transferencia de registros y la acción de secuenciamiento descrita en los comentarios de la derecha de la misma forma que se obtuvo la Tabla 10-15.

□ **TABLA 10-15**
Tabla de estados para ilustrar las instrucciones de tres ciclos o más

Estado	Entradas		Siguiente estado	Salidas												Comentarios
	Opcode	VCNZ		IL	PS	DX	AX	BX	MB	FS	MD	RW	MM	MW		
EX0	0010001	XXXX	EX1	0	00	1000	0XXX	XXXX	X	0000	1	1	X	0	LRI	$R8 \leftarrow M[R[SA], \rightarrow$ EX1
EX1	0010001	XXXX	INF	0	01	0XXX	1000	XXXX	X	0000	1	1	X	0	LRI	$R[DR] \leftarrow M[R8], \rightarrow$ INF*
EX0	0001101	XXX0	EX1	0	00	1000	0XXX	XXXX	X	0000	0	1	X	0	SRM	$R8 \leftarrow R[SA], \overline{Z}: \rightarrow$ EX1
EX0	0001101	XXX1	INF	0	01	1000	0XXX	XXXX	X	0000	0	1	X	0	SRM	$R8 \leftarrow R[SA], Z: \rightarrow$ INF*
EX1	0001101	XXX0	EX2	0	00	1001	XXXX	XXXX	1	1100	0	1	X	0	SRM	$R9 \leftarrow$ zf OP, $\overline{Z}: \rightarrow$ EX2
EX1	0001101	XXX1	INF	0	01	1001	XXXX	XXXX	1	1100	0	1	X	0	SRM	$R9 \leftarrow$ zf OP, $Z: \rightarrow$ INF*
EX2	0001101	XXXX	EX3	0	00	1000	XXXX	1000	0	1101	0	1	X	0	SRM	$R8 \leftarrow$ sr $R8, \rightarrow$ EX3
EX3	0001101	XXX0	EX2	0	00	1001	1001	XXXX	X	0110	0	1	X	0	SRM	$R9 \leftarrow R9 - 1, \overline{Z}: \rightarrow$ EX2
EX3	0001101	XXX1	EX4	0	00	1001	1001	XXXX	X	0110	0	1	X	0	SRM	$R9 \leftarrow R9 - 1, Z: \rightarrow$ EX4
EX4	0001101	XXXX	INF	0	01	0XXX	1000	XXXX	X	0000	0	1	X	0	SRM	$R[DR] \leftarrow R8, \rightarrow$ INF*
EX0	0001110	XXX0	EX1	0	00	1000	0XXX	XXXX	X	0000	0	1	X	0	SLM	$R8 \leftarrow R[SA], \overline{Z}: \rightarrow$ EX1
EX0	0001110	XXX1	INF	0	00	1000	0XXX	XXXX	X	0000	0	1	X	0	SLM	$R8 \leftarrow R[SA], Z: \rightarrow$ INF*
EX1	0001110	XXX0	EX2	0	01	1001	XXXX	XXXX	1	1100	0	1	X	0	SLM	$R9 \leftarrow$ zf OP, $\overline{Z}: \rightarrow$ EX2
EX1	0001110	XXX1	INF	0	01	1001	XXXX	XXXX	1	1100	0	1	X	0	SLM	$R9 \leftarrow$ zf OP, $Z: \rightarrow$ INF*
EX2	0001110	XXXX	EX3	0	00	1000	XXXX	1000	0	1110	0	1	X	0	SLM	$R8 \leftarrow$ sl $R8, \rightarrow$ EX3
EX3	0001110	XXX0	EX2	0	00	1001	1001	XXXX	X	0110	0	1	X	0	SLM	$R9 \leftarrow R9 - 1, \overline{Z}: \rightarrow$ EX2
EX3	0001110	XXX1	EX4	0	00	1001	1001	XXXX	X	0110	0	1	X	0	SLM	$R9 \leftarrow R9 - 1, Z: \rightarrow$ EX4
EX4	0001110	XXXX	INF	0	01	0XXX	1000	XXXX	X	0000	0	1	X	0	SLM	$R[DR] \leftarrow R8, \rightarrow$ IF*

* Para esta combinación de estado y entrada, también ocurre que $PC \leftarrow PC + 1$.

La realización de las instrucciones LRI y SRM ilustra la flexibilidad conseguida utilizando control multiciclo. La realización de instrucciones adicionales se explora en los problemas del final del capítulo.

10-10 RESUMEN DEL CAPÍTULO

En la primera parte del capítulo se presentó el concepto de ruta de datos para el procesado en sistemas digitales. Entre los principales componentes de la ruta de datos están los bancos de registros, los buses, las unidades aritmético-lógicas (ALUs) y los desplazadores. Las palabras de control proporcionan un medio de organizar el control de las microoperaciones realizadas en la ruta de datos. Estos conceptos se combinaron con el concepto de ruta de datos, que sirve como base para explorar los procesadores en el resto del texto.

En la segunda parte del capítulo, se presentó el diseño del control en sistemas programados, examinando dos formas diferentes de unidades básicas de control para un procesador con una arquitectura sencilla. Presentamos el concepto de arquitecturas de conjunto de instrucciones y definimos los formatos de las instrucciones y las operaciones de un procesador sencillo. La primera forma de este procesador es capaz de ejecutar cualquier instrucción en un solo ciclo de reloj. Aparte de tener un contador de programa y su lógica, la unidad de control de este procesador está compuesta por un circuito decodificador combinacional.

Entre las limitaciones de procesador de un solo ciclo están la complejidad de las instrucciones que se pueden ejecutar en él, problemas con la interfaz con una sola memoria, y la relativamente baja frecuencia de reloj obtenida. Para solventar las dos primeras limitaciones, examinamos una versión multiciclo de un procesador sencillo en el que se usa una sola memoria y las instrucciones se llevan a cabo en dos fases distintas: acceso a la instrucción y ejecución de la instrucción. El problema que resta por resolver, referente a utilizar muchos ciclos de reloj, se resolverá en el Capítulo 12 presentando la rutas de datos y el control en *pipeline*.

REFERENCIAS

1. MANO, M. M.: *Computer Engineering: Hardware Design:* Englewood Cliffs, NJ: Prentice Hall, 1988.

2. MANO, M. M.: *Computer System Architecture*, 3rd Ed. Englewood Cliffs, NY: Prentice Hall, 1993.

3. PATTERSON, D. A., and J. L. HENNESSY: *Computer Organization and Design: The Hardware/Software Interface*, 2nd ed. San Francisco, CA: Morgan Kaufmann, 1998.

4. HENNESSY, J. L., and D. A. PATTERSON: *Computer Architecture: A Quantitative Approach*, 2nd ed. San Francisco, CA: Morgan Kaufmann, 1996.

PROBLEMAS

El signo (+) indica problemas más avanzados y el asterisco (*) indica que hay una solución disponible en la dirección de Internet: http://www.librosite.net/Mano.

10-1. Una ruta de datos similar a la de la Figura 10-1 tiene 128 registros. ¿Cuántas líneas de selección se necesitan para cada grupo de multiplexores y para el decodificador?

10-2. *Dada una ALU de 8 bits con salidas F_7 a F_0 y acarreos disponibles C_8 y C_7, muestre el dibujo que genera las señales para los cuatro bits de status N (signo), Z (cero), V (*overflow*) y C (acarreo).

10-3. *Diseñe un circuito aritmético con dos líneas de selección S_1 y S_0 y datos A y B de n bits. El circuito debe efectuar las ocho siguientes operaciones aritméticas junto con el acarreo de entrada C_{in}:

S_1	S_0	$C_{in} = 0$	$C_{in} = 1$
0	0	$F = A + B$ (suma)	$F = A + \bar{B}$ (resta $A - B$)
0	1	$F = \bar{A} + B$	$F = \bar{A} + B$ (resta $B - A$)
1	0	$F = A - 1$ (decremento)	$F = A + 1$ (incremento)
1	1	$F = \bar{A}$ (complemento a 1)	$F = \bar{A} + 1$ (complemento a 2)

Dibuje el diagrama lógico para los dos bits menos significativos del circuito lógico.

10-4. *Diseñe un circuito aritmético de 4 bits, con dos variables de selección S_1 y S_0, que efectúe las siguientes operaciones aritméticas:

S_1	S_0	$C_{in} = 0$	$C_{in} = 1$
0	0	$F = A + B$ (suma)	$F = A + B + 1$
0	1	$F = A$ (transferencia)	$F = A + 1$ (incremento)
1	0	$F = \bar{B}$ (complemento)	$F = \bar{B} + 1$ (negación)
1	1	$F = A + \bar{B}$	$F = \bar{A} + \bar{B} + 1$ (resta)

Dibuje el diagrama lógico de una etapa de un bit.

10-5. Las entradas X_i e Y_i de cada sumador completo de un circuito aritmético tiene una lógica digital que está especificada por las funciones booleanas

$$X_i = A_i \qquad Y_i = \bar{B}_i S + B_i \bar{C}_{in}$$

donde S es una variable de selección, C_{in} es la entrada de acarreo, A_i y B_i son las entradas de datos para la etapa i.

(a) Dibuje el diagrama lógico de un circuito de 4 bits, utilizando sumadores completos y multiplexores.

(b) Determine la operación aritmética realizada por cada una de las cuatro combinaciones de S y C_{in}: 00, 01, 10 y 11.

10-6. *Diseñe una etapa de un bit de un circuito digital que realiza las cuatro siguientes operaciones: OR exclusiva, NOR exclusiva, NOR y NAND sobre los operandos de los registros A y B y deje el resultado en el registro A. Utilice dos variables de selección.

(a) Utilizando un Mapa de Karnaugh, diseñe la lógica mínima de una etapa, y dibuje el diagrama lógico.

(b) Repita el apartado (a) probando diferentes asignaciones de los códigos de selección para las cuatro operaciones para ver si la lógica de las etapas se pueden simplificar aún más.

10-7. + Diseñe una ALU que efectúe las siguientes operaciones:

$A + B$	sl A
$A + \overline{B} + 1$	$A \vee B$
$\overline{B}$	$A \oplus B$
$\overline{B} + 1$	$A \wedge B$

Dé el resultado de su diseño como un diagrama lógico para una etapa de la ALU. Su diseño debería tener una entrada de acarreo entre etapas y tres bits de selección. Si tiene acceso a software de simplificación, aplíquelo al diseño para obtener una lógica reducida.

10-8. *Encuentre la salida Y del *barrel shifter* de la Figura 10-9 para cada uno de los siguientes grupos de bits aplicados a S_1, S_0, D_3, D_2, D_1 y D_0:

(a) 000101 **(b)** 010011
(c) 101010 **(d)** 111100

10-9. Especifique la palabra de control de 16 bits que se debe aplicar a la ruta de datos de la Figura 10-11 para realizar las siguientes microoperaciones:

(a) $R0 \leftarrow R1 + R7$ **(b)** $R7 \leftarrow 0$
(c) $R6 \leftarrow$ sl $R6$ **(d)** $R3 \leftarrow$ sr $R4$
(e) $R1 \leftarrow R7 + 1$ **(f)** $R2 \leftarrow R4 -$ Constant in
(g) $R1 \leftarrow R2 \oplus R3$ **(h)** $R5 \leftarrow$ Data in

10-10. *Dadas las siguientes palabras de control de 16 bits para la ruta de datos de la Figura 10-11, determine (a) la microoperación que se ejecuta y (b) el cambio del contenido de los registros para cada palabra de control. Suponga que los registros son de 8 bits y que, antes de la ejecución de una palabra de control, contienen el valor de su número (por ejemplo, el registro $R5$ contiene 05 en hexadecimal). Suponga que Constant tiene el valor 6 y Data in tiene el valor 1B, ambos en hexadecimal.

(a) 101 100 101 0 1000 0 1 **(d)** 101 000 000 0 0000 0 1
(b) 110 010 100 0 0101 0 1 **(e)** 100 100 000 1 1101 0 1
(c) 101 110 000 0 1100 0 1 **(f)** 011 000 000 0 0000 1 1

10-11. Dada la siguiente secuencia de palabras de control de 16 bits para la ruta de datos de la Figura 10-11 y los carácteres iniciales ASCII en ocho registros, simule la ruta de datos para determinar los caracteres alfanuméricos en los registros después de ejecutar la secuencia. El resultado es una palabra cifrada. ¿Cuál es?

011 011 001 0 0010 0 1	$R0$	00000000
100 100 001 0 1001 0 1	$R1$	00100000
101 101 001 0 1010 0 1	$R2$	01000100
001 001 000 0 1011 0 1	$R3$	01000111
001 001 000 0 0001 0 1	$R4$	01010100
110 110 001 0 0101 0 1	$R5$	01001100
111 111 001 0 0101 0 1	$R6$	01000001
001 111 000 0 0000 0 1	$R7$	01001001

10-12. Una ruta de datos tiene cinco componentes principales, nombrados de *A* a *E*, conectados formando un bucle desde el banco de registros a otro banco de registros similar al de la Figura 10-17. El retardo máximo de cada componente es: *A*, 2 ns; *B*, 1 ns; *C*, 3 ns; *D*, 4 ns y *E*, 4 ns.

(a) ¿Cuál es la frecuencia máxima de reloj que se puede utilizar en la ruta de datos?

(b) La ruta de datos se ha cambiado a otra en *pipeline* utilizando tres registros. ¿Cómo se deberían distribuir los componentes entre las etapas y cuál es la frecuencia de reloj máxima que se podría conseguir?

(c) Repita el apartado (b) para un *pipeline* de 4 etapas.

10-13. Un procesador tiene instrucciones de 32 bits repartidos en los siguientes campos: opcode, 6 bits; dos campos de registros, 6 bits cada uno; y un operando inmediato o campo de registro de 14 bits.

(a) ¿Cuál es el número máximo de operaciones que se pueden especificar?

(b) ¿Cuántos registros se pueden direccionar?

(c) ¿Cuál es el rango de los operandos inmediatos sin signo que se pueden utilizar?

(d) ¿Cuál es el rango de operandos inmediatos con signo que se pueden utilizar, suponiendo que el bit 13 es el bit de signo?

10-14. *Un procesador tiene una unidad de memoria con 32 instrucciones y un banco de registros con 32 registros. El conjunto de instrucciones está formado por 110 operaciones diferentes. Hay un solo tipo de formato para las instrucciones, con una parte para el opcode, una para la dirección para el banco de registros y otra para un operando inmediato. Cada instrucción se almacena en una palabra de la memoria.

(a) ¿Cuántos bits son necesarios para la parte del opcode?

(b) ¿Cuántos bits quedan para la parte del operando inmediato de la instrucción?

(c) Si el operando inmediato se utiliza como una dirección de memoria sin signo, ¿cuál es el número máximo de palabras de memoria que se pueden direccionar?

(d) ¿Cuál es el valor algebraico más grande y más pequeño en complemento a 2 que se puede utilizar como operando inmediato?

10-15. Un procesador tiene instrucciones de 32 bits. Hay varios formatos diferentes y el número de bits de los opcodes varían dependiendo de los bits que necesitan los otros campos. Si el primer bit del opcode es 0, entonces hay cuatro bits para el opcode. Si el primer bit del opcode es 0, el opcode tiene 6 bits. Si el primer bit de opcode es 1 y el segundo es 1, el opcode tiene 8 bits. ¿Cuántos opcodes hay disponibles para el procesador?

10-16. El procesador de la Figura 10-15 ejecuta las cinco instrucciones descritas, mediante transferencias de registros, en la siguiente tabla:

(a) Complete la tabla dando las salidas del decodificador de instrucciones binario de la Figura 10-16 durante la ejecución de cada una de las instrucciones:

Instrucción-transferencia de registros	DA	AA	BA	MB	FS	MD	RW	MW	PL	JB
$R[0] \leftarrow R[7] \oplus R[3]$										
$R[1] \leftarrow M[R[4]]$										
$R[2] \leftarrow R[5] + 2$										
$R[3] \leftarrow$ sl $R[6]$										
if $(R[4] = 0)$ $PC \leftarrow PC +$ se PC else $PC \leftarrow PC + 1$										

(b) Complete la siguiente tabla, dando la instrucción en binario para un procesador de un solo ciclo que ejecuta las transferencias de registros (si algún campo no se utiliza, déle el valor 0):

Instrucción-transferencia de registros	Opcode	DR	SA	SB u operando
$R[0] \leftarrow$ sr $R[7]$				
$R[1] \leftarrow M[R[6]]$				
$R[2] \leftarrow R[5] + 4$				
$R[3] \leftarrow R[4] \oplus R[3]$				
$R[4] \leftarrow R[2] - R[1]$				

10-17. Utilizando la información de la tabla de verdad de la Tabla 10-10, verifique que el diseño de las salidas para un solo bit del decodificador de la Figura 10-16 es correcto.

10-18. Simule manualmente el procesador de un solo ciclo de la Figura 10-15 para la siguiente secuencia de instrucciones, suponiendo que cada registro contiene inicialmente su índice como dato (es decir, *R*0 contiene 0, *R*1 contiene 1, etc.):

```
SUB R0, R1, R2
SUB R3, R4, R5
SUB R6, R7, R0
SUB R0, R0, R3
SUB R0, R0, R6
ST R7, R0
LD R7, R6
ADI R0, R6, 0
ADI R3, R6, 3
```

Indique:

(a) El valor binario de la instrucción junto con los resultados en la misma línea de la instrucción.
(b) El contenido de cualquier registro que haya sido cambiado por la instrucción, o la posición y contenido de la memoria que haya sido cambiado por la instrucción en la siguiente línea de la instrucción en curso. Los resultados se colocan de esta forma ya que los valores nuevos no aparecen en un registro o memoria, debido a la ejecución de una instrucción, hasta después que haya ocurrido un flanco de subida de la señal de reloj.

10-19. Indique una instrucción para un procesador de un solo ciclo de reloj que ponga a cero el registro *R*4 y actualice los bits de status *Z* y *N* cuando se transfiera el valor 0 a *R*4. (*Sugerencia:* utilice una OR exclusiva.) Examinado la lógica de la ALU, determine los valores de los bits de status *V* y *C*.

10-20. Indique las entradas de la tabla de estados de la lógica de control para un procesador multiciclo (*véase* la Tabla 10-15) que ejecuta las siguientes sentencias de transferencia de registro. Suponga que, en todos los casos, el estado actual es EX0 y el código de operación es 0010001.

(a) $R3 \leftarrow R1 - R2$, *EX*1, suponga que DR = 3, SA = 1 y SB = 2.
(b) $R8 \leftarrow$ sr $R8, \rightarrow INF$, suponga que DR = 5 y SB = 5
(c) if ($N = 0$) then ($PC \rightarrow PC +$ se, *INF*) else ($PC \rightarrow PC + 1$, *INF*)
(d) $R6 \leftarrow R6$, $C \leftarrow 0$, $\rightarrow INF$, suponga que DR = SA = 6.

10-21. Simule manualmente la instrucción SRM en un procesador multiciclo para el operando 0001001101111000 para OP = 6.

10-22. Se define una nueva instrucción para el procesador multiciclo con opcode 0010001. La instrucción efectúa la transferencia de registro

$$R[\text{DR}] \leftarrow R[\text{SB}] + M[R[\text{SA}]]$$

Encuentre el diagrama ASM que realice la instrucción, suponiendo que el opcode es 0010001. Haga la parte de la tabla de control de estados que lleva a cabo esta instrucción.

10-23. Repita el Problema 10-22 para las instrucciones: suma y comprueba (*add and check*) OV (AOV), descrita mediante la transferencia de registros

$$R[\text{DR}] \leftarrow R[\text{SA}] + R[\text{SB}],\ V\text{: } R8 \leftarrow 1,\ \bar{V}\text{: } R8 \rightarrow 0$$

y bifurcación sobre *overflow* (BRV), descrita mediante la transferencia de registros

$$R8 \leftarrow R8,\ V\text{: } PC \leftarrow PC + \text{se AD},\ \bar{V}\text{: } PC \leftarrow PC + 1$$

El opcode de AOV es 1000101 y, para BRV, es 1000110. Véase que el registro *R*8 se utiliza como registro de «status» que almacena el resultado de *overflow* de la anterior operación. El resto de valores de *N*, *Z*, *C* y *V* se podrían almacenar en *R*8 dando una información completa de status de la anterior operación lógica o aritmética.

10-24. + Se define una nueva instrucción para un procesador multiciclo. La instrucción compara dos números enteros sin signo almacenados en los registros R[SA] y R[SB]. Si los enteros son iguales, entonces el bit 0 de R[DR] se pone a 1. Si R[SA] es mayor que R[SB], entonces el bit 1 de R[DR] se pone a 1. En el resto de casos, los bits 0 y 1 son 0. El resto de bits de R[DR] tienen valor 0. Encuentre el diagrama ASM para ejecutar la instrucción, suponiendo que el opcode es 0010001. Haga la parte de la tabla de control de estados que lleva a cabo esta instrucción.

CAPÍTULO

11

ARQUITECTURA DE CONJUNTO DE INTRUCCIONES

Hasta este punto, la mayor parte de lo que hemos estudiado está enfocado al diseño de sistemas digitales, utilizando como ejemplos componentes de procesadores. En este capítulo, el material estudiado pasa a ser claramente más especializado, relacionándose con la arquitectura de conjunto de instrucciones de procesadores de propósito general. Examinaremos las operaciones que realizan las instrucciones, enfocándonos en particular en cómo se obtienen los operandos y dónde se almacenan los resultados. En nuestro estudio contrastaremos dos tipos de arquitecturas distintas: procesadores de conjunto reducido de instrucciones (RISC, del inglés *Reduced Instruction Set Computers*) y los procesadores de conjunto de instrucciones complejo (CISC, del inglés *Complex Instruction Set Computers*). Clasificaremos las instrucciones básicas en tres categorías: transferencia de datos, manipulación de datos y control de programa. En cada una de estas categorías detallaremos las instrucciones básicas típicas.

Para aclarar este cambio de enfoque, se han matizado las partes de propósito general del procesador de genérico presentado al comienzo del Capítulo 1, que incluye la unidad central de procesamiento (CPU) y la unidad de punto flotante (FPU). Además, como puede estar presente un pequeño microprocesador de propósito general para controlar un teclado y las funciones de un monitor, hemos matizado estos componentes ligeramente. Aparte del direccionamiento utilizado para acceder a la memoria y a los componentes de entrada/salida (E/S), los conceptos estudiados se aplican después a otras áreas del procesador. Sin embargo, aparecen con más frecuencia, pequeñas CPUs en los dispositivos de E/S, dando un cambio en el panorama del papel que juegan las arquitecturas de conjunto de instrucciones de propósito general en el procesador genérico.

11-1 CONCEPTOS DE LA ARQUITECTURA DE PROCESADORES

Al lenguaje binario en el que las instrucciones se definen y se almacenan en la memoria se le llama *código máquina*. El lenguaje que sustituye a los códigos de operación binarios y las direcciones con nombres simbólicos y aquello que proporciona otras características útiles al programador se le llama *lenguaje ensamblador*. La estructura lógica de los procesadores se describe normalmente en los manuales de referencia del lenguaje ensamblador. Tales manuales describen los diversos elementos internos del procesador que son de interés para el programador, como son los registros del procesador. Los manuales enumeran todas las instrucciones que puede realizar el hardware, especifican los nombres simbólicos y el formato del código binario de las instrucciones y proporcionan una definición precisa de cada instrucción. En el pasado, esta información representó la *arquitectura* de un procesador. Un procesador estaba compuesto por su arquitectura, y una *implementación* específica de tal arquitectura. Su diseño estaba separado en dos partes: la organización y el hardware. La *organización* esta formada por estructuras como las rutas de datos, unidades de control, memoria y los buses que las interconectan. El *hardware* hace referencia a la lógica, la tecnología electrónica empleada y diversos aspectos físicos del diseño del procesador. Según el diseño de procesadores necesitaban más y más rendimiento, y la tendencia de que la mayor parte del procesador estuviese en un solo circuito integrado, la relación entre arquitectura, organización y hardware llegó a ser tan estrecha que fue necesario tener un punto de vista más integrador. De acuerdo con este nuevo punto de vista, la arquitectura anteriormente definida se llama más estrictamente *arquitectura de conjunto de instrucciones* (ISA del inglés *Instruction Set Architecture*), y el término *arquitectura* se usa para abarcar a todo el procesador, incluyendo la arquitectura de conjunto de instrucciones, organización y el hardware. Esta visión unificada permite hacer consideraciones en el diseño inteligente que son evidentes en un proceso de diseño tan fuertemente interrelacionado. Estas consideraciones tienen el potencial de generar mejores diseños de los procesadores. En este capítulo, nos centraremos en la arquitectura de conjunto de instrucciones. En el siguiente, veremos dos arquitecturas de conjunto de instrucciones diferentes, con un enfoque en la realización de dos organizaciones muy diferentes.

Un procesador tiene habitualmente gran variedad de instrucciones y formatos de éstas. La función de la unidad de control es descodificar cada instrucción y proporcionar las señales de control necesarias para ejecutarlas. En la Sección 10-7 se presentaron ejemplos sencillos de instrucciones y formatos de instrucciones. Extenderemos esa presentación introduciendo las instrucciones típicas que se encuentran en los procesadores comerciales de propósito general. También averiguaremos los diversos formatos de instrucciones que se pueden encontrar típicamente en un procesador, haciendo énfasis en el direccionamiento de los operandos. El formato de una instrucción se representa como una caja rectangular simbolizando los bits de la instrucción en binario. Los bits se dividen en grupos llamados *campos*. A continuación se comentan los campos típicos que se encuentran en los formatos de las instrucciones:

1. Un *campo de código de operaciones*, que especifica la operación a realizar.
2. Un *campo de direcciones*, que proporciona direcciones de la memoria o direcciones para un registro del procesador.
3. Un *campo de modo*, que especifica la forma en que se interpreta el campo de direcciones.

En ciertas circunstancias, se emplean a veces campos especiales, por ejemplo, un campo que especifica el número de posiciones de desplazamiento de un dato en una instrucción de desplazamiento, o un campo de operando para las instrucciones de operando inmediato.

Ciclo de operación básico de un procesador

Para entender los diversos conceptos de direccionamiento que se presentan en las dos siguientes secciones necesitamos comprender el ciclo básico de un procesador. La unidad de control de un procesador se diseña para ejecutar cada una de las instrucciones de un programa efectuando la siguiente secuencia de pasos:

1. Traer la instrucción de la memoria a un registro de control.
2. Descodificar la instrucción.
3. Localizar los operandos utilizados por la instrucción.
4. Traer los operandos de la memoria (si es necesario)
5. Ejecutar la operación en los registros del procesador.
6. Almacenar el resultado en el lugar adecuado
7. Regresar al paso 1 para traer la siguiente instrucción.

Como se explicó en la Sección 10-7, en el procesador hay un registro que se llama contador de programa (*PC*) que sigue la pista de las instrucciones del programa almacenado en la memoria. El *PC* mantiene la dirección de la instrucción que se va a ejecutar a continuación y se incrementa en uno cada vez que se lee una palabra del programa almacenado en la memoria. La descodificación hecha en el paso 2 determina la operación a ejecutar y el modo de direccionamiento de la instrucción. En el paso 3, los operandos se localizan según el modo de direccionamiento y el campo de direcciones de la instrucción. El procesador ejecuta la instrucción, almacena el resultado y regresa al paso 1 para coger la siguiente instrucción de la secuencia.

Conjunto de registros

El conjunto de registros está formado por todos los registros de la CPU accesibles por el programador. Estos registros son, típicamente, los que se mencionan en los manuales de referencia de programación de los lenguajes ensamblador. En las CPUs sencillas, como ya hemos comentado, el conjunto de registros está formado por una parte accesible del banco de registros y el *PC*. Las CPUs también pueden tener otros registros, como el registro de instrucciones, registros pertenecientes al banco de registros que sólo son accesibles a los microprogramas, y los registros de *pipeline*. Sin embargo, estos registros no están accesibles al programador y, por tanto, no forman parte del conjunto de registros que representan a la información almacenada en la CPU a la que las instrucciones puedan acceder. De esta forma, el conjunto de registros tiene una gran influencia en la arquitectura del conjunto de instrucciones.

El conjunto de registros para una CPU real puede llegar a ser bastante complejo. Para la presentación de este capítulo, añadiremos dos registros al conjunto que hemos utilizado con anterioridad: el *registro de status de procesador* (*PSR*, del inglés *processor status register*) y el *puntero de pila* (*SP*, del inglés *stack pointer*). El registro de status del procesador contiene flip-flops que se pueden cambiar individualmente por los valores de status *C*, *N*, *V* y *Z* de la ALU. Estos bits de status almacenados se utilizan para tomar decisiones que determinan el flujo del programa, teniendo en cuenta los resultados de la ALU o en el contenido de los registros. A los bits almacenados en el registro de status del procesador se les denomina *códigos de condición* o *banderas* (en inglés *flags*). Se verán otros bits del *PSR* cuando se hayan cubierto los conceptos asociados a este capítulo.

11-2 DIRECCIONAMIENTO DE LOS OPERANDOS

Consideremos una instrucción como ADD, que especifica la suma de dos operandos y produce un resultado. Suponga que, en este caso, el resultado de la suma se trata de otro operando. Por tanto, la instrucción ADD tiene tres operandos, los dos sumandos y el resultado. Un operando, residente en la memoria, se especifica por su dirección. Un operando en un registro del procesador se especifica por una dirección de registro, que es un código binario de n bits que especifica uno de los 2^n registros del banco de registros. De esta manera, un procesador con 16 registros de procesador, llamémosles de $R0$ a $R15$, tiene en su instrucción uno o más campos de dirección de registro con cuatro bits. Por ejemplo, el código binario 0101 indica el registro $R5$.

Sin embargo, algunos operandos no tienen una dirección explicita pues está incluida por el código de operación de la instrucción o por una dirección asignada a uno de los otros operando. En tal caso, diremos que el operando tiene una *operación implícita*. Si la dirección es implícita, entonces no hay necesidad de un campo de direcciones de memoria o de registro para el operando de la instrucción. Por otra parte, si el operando tiene una dirección en la instrucción, diremos que el operando se direcciona explícitamente o que tiene una *dirección explícita*.

El número de operandos direccionados explícitamente para una operación de manipulación de datos, como en la instrucción ADD, es un factor importante a la hora de definir la arquitectura de conjunto de instrucciones de un procesador. Un factor adicional es el número de operandos que se pueden direccionar explícitamente en la memoria mediante una instrucción. Estos dos factores son tan importantes a la hora de definir la naturaleza de las instrucciones que representan un medio de distinguir las diferentes arquitecturas de conjunto de instrucciones. Además, definen la longitud de las instrucciones del procesador.

Empezaremos ilustrando programas sencillos con diferente número de operandos direccionados explícitamente por instrucción. Puesto que cada operando direccionado explícitamente tiene hasta tres direcciones de memoria o de registro por instrucción, etiquetaremos las instrucciones como instrucciones de tres, dos, una o cero direcciones. Véase que, de los tres operandos necesarios para una instrucción como ADD, las direcciones de todos los operandos que no tienen una dirección en la instrucción están implícitas.

Para ilustrar cómo afecta el número de operando en los programas de procesador, evaluaremos la siguiente sentencia aritmética:

$$X = (A + B)(C + D)$$

usando instrucciones de tres, dos, una y cero direcciones. Supondremos que los operandos están en las direcciones de memorias representadas por las letras A, B, C y D, y no deben ser cambiados por el programa. El resultado se almacena en memoria, en una posición con dirección X. Las operaciones aritméticas que se pueden utilizar son la suma, la substracción y la multiplicación, designadas por ADD, SUB y MUL, respectivamente. Además, las tres operaciones que se necesitan para transferir los datos durante la ejecución son mover, cargar y almacenar, designadas por MOVE, LD y ST, respectivamente. LD mueve un operando de la memoria a un registro y ST de un registro a la memoria. Dependiendo de las direcciones permitidas, MOVE puede transferir datos entre registros, entre posiciones de memoria, de la memoria a un registro o de un registro a la memoria.

Instrucciones de tres direcciones

A continuación se muestra un programa que calcula $X = (A + B)(C + D)$ utilizando instrucciones de tres direcciones (para cada instrucción se muestra una sentencia de transferencia de registro):

ADD T1, A, B	$M[T1] \leftarrow M[A] + M[B]$
ADD T2, C, D	$M[T2] \leftarrow M[C] + M[\text{D}]$
MUL X, T1, T2	$M[X] \leftarrow M[T1] \times M[T2]$

El símbolo $M[A]$ indica el operando almacenado en la memoria en la dirección representada por A. El símbolo $\times$ indica multiplicación. $T1$ y $T2$ son las posiciones de almacenamiento de la memoria.

El mismo programa puede usar registros como posiciones de almacenamiento temporal:

ADD R1, A, B	$R1 \leftarrow M[A] + M[B]$
ADD R2, C, D	$R2 \leftarrow M[C] + M[\text{D}]$
MUL X, R1, R2	$M[X] \leftarrow R1 \times R2$

El uso de los registros reduce el tiempo de los accesos requeridos entre 5 y 9 veces. Una ventaja del formato de tres direcciones es que abrevia los programas de evaluación de expresiones. Una desventaja es que el código binario de la instrucción necesita más bits para especificar las tres direcciones, en particular si hay direcciones de memoria.

Instrucciones de dos direcciones

En las instrucciones de dos direcciones, cada campo de dirección puede especificar una dirección de registro o una dirección de memoria. La primera dirección del operando que aparece en la instrucción simbólica también sirve como dirección implícita a la que el resultado de la operación se transfiere. El programa es como sigue:

MOVE T1, A	$M[T1] \leftarrow M[A]$
ADD T1, B	$M[T1] \leftarrow M[T1] + M[B]$
MOVE X, C	$M[X] \leftarrow M[C]$
ADD X, D	$M[X] \leftarrow M[X] + M[D]$
MUL X, T1	$M[X] \leftarrow M[X] \times M[T1]$

Si hay disponible un registro de almacenamiento temporal, $R1$, se puede reemplazar $T1$. Véase que este programa tiene cinco instrucciones en lugar de las tres usadas en el programa con instrucciones de tres direcciones.

Instrucciones de una dirección

Para realizar instrucciones como ADD, un procesador con instrucciones con una sola dirección utiliza direcciones implícitas, como un registro llamado *acumulador*, ACC, para obtener uno de

los operando y como posición para guardar el resultado. El programa que evalúa la sentencia aritmética es el siguiente:

LD	A	$ACC \leftarrow M[A]$
ADD	B	$ACC \leftarrow ACC + M[B]$
ST	X	$M[X] \leftarrow ACC$
LD	C	$ACC \leftarrow M[C]$
ADD	D	$ACC \leftarrow ACC + M[D]$
MUL	X	$ACC \leftarrow ACC \times M[X]$
ST	X	$M[X] \leftarrow ACC$

Todas las operaciones se han realizado entre un operando almacenado en el registro *ACC* y un operando de la memoria. En este caso, el número de instrucciones se ha incrementado a 7 y los accesos a memoria se elevan también a siete.

Instrucciones con cero direcciones

Para realizar una instrucción ADD con cero direcciones, las tres direcciones de la instrucción deben ser implícitas. Una forma convencional de lograr este objetivo es utilizar estructuras llamadas *pilas* (*stack* en inglés), que almacenan información de forma que el último elemento almacenado es el primero en recuperarse. Debido a la naturaleza de esta pila, el último en entrar es el primero en salir, se le llama *pila LIFO* (del inglés, *last in, first out*). El funcionamiento de un procesador con pila es análogo al de una pila de bandejas o platos en la que la bandeja colocada encima de la pila es la primera que se coge. Las operaciones de manipulación de datos como la instrucción ADD se realizan sobre la pila. A la palabra de la parte superior de la fila la llamaremos TOS (del inglés *Top of Stack*). A la palabra debajo de ésta la llaremos TOS_{-1}. Cuando se usan uno o más operandos en una operación, estos se eliminan de la pila. La palabra por debajo pasa entonces a ser la nueva TOS. Cuando se genera una palabra como resultado, se coloca en la pila pasando a ser la nueva TOS. De esta forma, TOS y algunas palabras ubicadas debajo de ella son las direcciones implícitas de los operandos y TOS es la dirección implícita del resultado. Por ejemplo, la instrucción que especifica una suma es simplemente

ADD

El resultado de efectuar la transferencia de registros es $TOS \leftarrow TOS + TOS_{-1}$. Así vemos que en las instrucciones de manipulación de datos en una arquitectura con pila no utilizan ni registros ni direcciones de registros. Sin embargo, el direccionamiento de la memoria se utiliza en dichas arquitecturas para la transferencia de datos. Por ejemplo, la instrucción

PUSH X

da lugar a una transferencia de una palabra de la dirección X de la memoria a la posición superior de la pila. La siguiente operación

POP X

da lugar a la transferencia de una palabra de la posición superior de la pila a la dirección X de la memoria. El programa que evalúa la sentencia aritmética para instrucciones de cero direcciones es el siguiente:

PUSH	A	$TOS \leftarrow M[A]$
PUSH	B	$TOS \leftarrow M[B]$
ADD		$TOS \leftarrow TOS + TOS_{-1}$
PUSH	C	$TOS \leftarrow M[C]$
PUSH	D	$TOS \leftarrow M[D]$
ADD		$TOS \leftarrow TOS + TOS_{-1}$
MUL		$TOS \leftarrow TOS \times TOS_{-1}$
POP	X	$M[X] \leftarrow TOS$

Este programa necesita ocho instrucciones, una más que en el programa anterior. Sin embargo, utiliza direcciones de posiciones de memoria o registros solo para las instrucciones PUSH y POP y no para ejecutar instrucciones de manipulación de los datos involucrados en las instrucciones ADD y MUL.

Arquitecturas de direccionamiento

Los programas que se acaban de presentar cambian si se restringe el número de direcciones de memoria de la instrucción o si las direcciones de memoria se restringen a algunas instrucciones concretas. Estas restricciones, combinadas con el número de operandos direccionados, definen la arquitectura de direccionamiento. Podemos ilustrar tales arquitecturas evaluando una sentencia aritmética en una arquitectura de tres direcciones donde todas las instrucciones tienen acceso a la memoria. A dicho modelo de direccionamiento se le llama *arquitectura memoria a memoria*. Esta arquitectura solamente tiene registros de control, como el contador de programa de una CPU. Todos los operandos proceden directamente de la memoria y todos los resultados se mandan directamente a la memoria. Los formatos de las instrucciones de transferencia y manipulación de los datos contienen entre uno y tres campos de direcciones. En el ejemplo anterior, se necesitan tres instrucciones, pero si aparece una nueva palabra debe aparecer en la instrucción para cada dirección de memoria, entonces se necesitan cuatro lecturas de memoria para traer cada una de las instrucciones. Incluyendo la extracción de los operandos y el almacenamiento de los resultados, el programa que realiza la suma podría necesitar 21 accesos a la memoria. Si los accesos de memoria necesitan más de un ciclo de reloj, el tiempo de ejecución podría pasar de 21 ciclos de reloj. De forma que, aunque el número de instrucciones sea bajo, el tiempo de ejecución puede ser alto. Además, dar la posibilidad de que todas las operaciones puedan acceder a la memoria, incrementa la complejidad de las estructuras de control dando como resultado que el periodo de la señal de reloj tenga que ser más largo, bajando, por tanto, la frecuencia de funcionamiento. Por esto, la arquitectura memoria a memoria no se usa, generalmente, en los diseños actuales.

Por el contrario, las *arquitecturas de registro a registros* con tres direcciones o *arquitecturas carga/almacenamiento*, que sólo permiten una dirección de memoria y restringe su uso a las instrucciones de carga y almacenamiento, son típicas en el diseño de los procesadores moder-

nos. Dicha arquitectura necesita un banco de registros con un tamaño adecuado, ya que todas las instrucciones de manipulación de datos utilizan los registros como operandos. Con esta arquitectura, el programa que evalúa la sentencia aritmética de ejemplo es el siguiente:

```
LD      R1, A          R ← M[A]
LD      R2, B          R2 ← M[B]
ADD     R3, R1, R2     R3 ← R1 + R2
LD      R1, C          R1 ← M[C]
LD      R2, D          R2 ← M[D]
ADD     R1, R1, R2     R1 ← R1 + R2
MUL     R1, R1, R3     R1 ← R1 × R3
ST      X, R1          M[X] ← R1
```

Véase que el número de instrucciones pasa a ocho en lugar de las tres que se utilizaron en el caso de la arquitectura memoria a memoria. Además, las operaciones son las mismas que las del caso de la arquitectura con pila, excepto el uso de direcciones para los registros. Con el uso de los registros, el número de accesos a memoria para obtener las instrucciones se reduce de 21 a 18. Si las direcciones se obtienen de los registros en lugar de la memoria, como se estudia en la siguiente sección, este número se puede reducir aún más.

Otras variantes de las dos arquitectura de direccionamiento anteriores incluyen instrucciones de tres direcciones e instrucciones de dos direcciones con una o dos direcciones de memoria. La longitud del programa y el número de accesos a memoria tienden a estar en un punto intermedio de las dos arquitecturas anteriores. Un ejemplo de una instrucción de dos direcciones permitiendo una sola dirección de memoria es:

```
ADD     R1, A          R1 ← R1 + M[A]
```

Este tipo de arquitectura se llama arquitectura de *memoria-registro* y predomina entre las arquitecturas de conjunto de instrucciones habituales, principalmente por dar compatibilidad con programas antiguos que utilizan una arquitectura concreta.

El programa con instrucciones con una dirección, ilustrado anteriormente, tiene una *arquitectura de un solo acumulador*. Como esta arquitectura no tiene banco de registros, su única dirección es para acceder a la memoria. Esto requiere 21 accesos a la memoria para evaluar la sentencia aritmética del ejemplo. En programas más complejos, se suele necesitar un notable aumento de acceso de memoria. Debido al gran número de accesos a la memoria, esta arquitectura resulta ineficiente y, por tanto, se restringe al uso de aplicaciones sencillas y de bajo coste, que no requieren un alto rendimiento.

El caso de las instrucciones con cero direcciones que utilizan una pila, se mantiene el concepto de *arquitectura de pila*. Las instrucciones de manipulación de datos, como ADD, no utilizan direcciones, ya que se realizan utilizando pocos elementos de la pila. Para la transferencia de datos se utilizaron las operaciones de carga y almacenamiento de una sola dirección de memoria, como se mostró en el programa del ejemplo que evalúa una sentencia aritmética. Al estar gran parte de la pila ubicada en la memoria, puede ser necesario efectuar uno o más accesos de memoria ocultos para cada operación con la pila. Las arquitecturas registro-registro y carga/almacenamiento han dado lugar a importantes progresos en el rendimiento, el gran número de

accesos a memoria de las arquitecturas basadas en pilas las han hecho poco atractivas. Sin embargo, arquitecturas de pila recientes han comenzado a apropiarse de los avances tecnológicos que han surgido en las otras arquitecturas. Estas nuevas arquitecturas almacenan un importante número de posiciones de pila en el procesador y manejan transferencias entre estas posiciones y la memoria de forma transparente. Las arquitecturas basadas en pilas son particularmente útiles para la rápida interpretación de lenguajes de programación de alto nivel, en los que la representación intermedia del código utiliza operaciones con la pila. Las arquitecturas con pilas son compatibles con un método muy eficiente de expresar procesos que usan notación postfija (en inglés *postfix*) en lugar de la notación tradicional infijo (en inglés *infix*) a la que estamos acostumbrados. La expresión infijo

$$(A + B) \times C + (D \times E)$$

con los operadores entre los operandos se puede escribir como una expresión postfijo

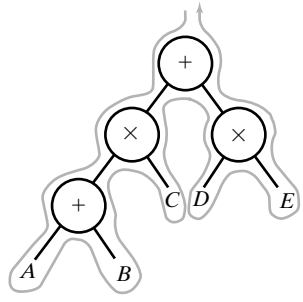
$$A\ B + C \times D\ E \times +$$

La notación postfijo también se llama notación polaca inversa (NPI), nombrada así por el matemático Jan Lukasiewicz, quien propuso la notación prefijo (la contraria a la postfijo).

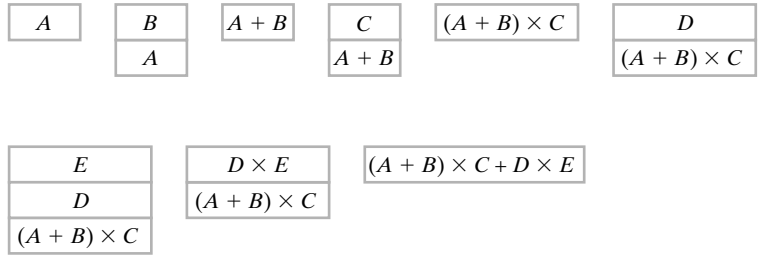
La conversión de $(A + B) \times C + (D \times E)$ a NPI se puede conseguir gráficamente, según se muestra en la Figura 11-1. Cuando el camino que atraviesa el gráfico pasa por una variable, ésta se introduce en la expresión NPI.

Es muy sencillo realizar un programa para evaluar una expresión en NPI. Siempre que se encuentra una variable se coloca en la pila. Siempre que se encuentra una operación, ésta se ejecuta usando la dirección implícita TOS, o se direcciona TOS y TOS_{-1}, colocándose el resultado en la nueva TOS. Un ejemplo de programa para una expresión en NPI es:

```
PUSH A
PUSH B
ADD
PUSH C
MUL
PUSH D
PUSH E
MUL
ADD
```



□ **FIGURA 11-1**
Gráfico de ejemplo de conversión de infijo a NPI



□ **FIGURA 11-2**
Actividad de la pila en la ejecución del ejemplo del programa

La ejecución del programa se ilustra en la Figura 11-2, mediante los estados sucesivos de la pila. Cuando un operando se coloca en la pila, el contenido de la pila se desplaza una posición hacia abajo. Cuando se ejecuta una operación, el operando en la TOP se saca de la pila y se almacena temporalmente en un registro. Se realiza la operación sobre el operando almacenado y el nuevo operando de la TOS, y el resultado reemplaza al operando de la TOS.

11-3 MODOS DE DIRECCIONAMIENTO

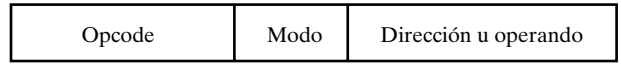
El campo de operación de una instrucción especifica la operación a ejecutar. Esta operación debe ejecutarse sobre el dato almacenado en los registros del procesador o en las palabras de la memoria. Cómo se seleccionan los operandos durante la ejecución del programa depende del modo de direccionamiento de la instrucción. El modo de direccionamiento especifica una regla para interpretar o modificar el campo de direcciones de la instrucción antes de que se haga realmente referencia al operando. A la dirección del operando generada mediante la aplicación de esa regla se le llama *dirección efectiva*. Los procesadores utilizan técnicas de modo de direccionamiento para ajustarse a las siguientes características:

1. Proporcionar flexibilidad al usuario en la programación mediante punteros a la memoria, contadores para el control de bucles, indexar datos y reubicar programas.
2. Reducir el número de bits de los campos de direcciones de la instrucción.

Disponer de varios modos de direccionamientos proporciona al programador experimentado la posibilidad de escribir programas que requieran pocas instrucciones. Sin embargo, el efecto en el trasiego de datos y el tiempo de ejecución debe sopesarse cuidadosamente. Por ejemplo, la presencia de modos de direccionamiento más complejos puede, en realidad, dar lugar a un bajo movimiento de datos y a un tiempo de ejecución más largo. Además, muchos códigos máquina de programas se generan por compiladores que, con frecuencia, no utilizan los modos de direccionamientos complejos de forma efectiva.

En algunos procesadores, el modo de direccionamiento de la instrucción se especifica mediante distinto código binario. Otros procesadores utilizan un código binario común tanto para operaciones y modos de direccionamiento de la instrucción. Las instrucciones se pueden definir con variedad de modos de direccionamiento y, a veces, se combinan varios modos de direccionamiento en una instrucción.

En la Figura 11-3 se muestra el formato de una instrucción con un campo de modo de direccionamiento distinto. El código de operación especifica la operación a realizar. El campo de modo se utiliza para localizar los operandos necesarios para la operación. Puede o no haber un campo de dirección en la instrucción. Si existe un campo de dirección, puede ser para una direc-



□ **FIGURA 11-3**
Formato de una instrucción con campo de modo de direccionamiento

ción de memoria o para un registro del procesador. Además, según se estudió en la sección anterior, la instrucción puede tener más de un campo de direcciones. En tal caso, cada campo de dirección se asocia con su modo particular de direccionamiento.

Modo implícito

Aunque la mayoría de los modos de direccionamiento modifica el campo de direcciones de la instrucción, hay un modo que no necesita campo de direcciones: el modo implícito. En este modo, el operando se especifica implícitamente en el código de operaciones. El modo implícito, que proporciona la ubicación para las operaciones con dos operando más resultado, es donde se combinan en una instrucción menos de tres direcciones. Por ejemplo, la instrucción «complementar el acumulador» tiene un modo implícito ya que el operando del registro acumulador queda implícito en la definición de la instrucción. De hecho, cualquier instrucción que utiliza el acumulador sin un segundo operando es una instrucción con modo implícito de direccionamiento. Por ejemplo, las instrucciones de manipulación de datos en la pila del procesador, como ADD, son instrucciones con modo implícito, ya que los operandos están implícitamente en la posición superior de la pila.

Modo inmediato

En el modo inmediato, el operando se especifica en la propia instrucción. En otras palabras, una instrucción de modo inmediato tiene un campo de operando en lugar de un campo de direcciones. El campo de operando contiene el operando a utilizar junto con la operación que se especifica en la instrucción. Las instrucciones con modo inmediato son muy utilizadas, por ejemplo, para inicializar registros con un valor constante.

Modos registro y registro indirecto

Anteriormente, se comentó que el campo de direcciones de una instrucción puede especificar tanto una posición de memoria como un registro del procesador. Cuando el campo de direcciones especifica un registros del procesador, se dice que está en modo registro. En este modo los operando están en los registros que están dentro del procesador. Un registro en particular se selecciona mediante un campo de dirección de registro dentro del formato de la instrucción.

En el modo registro indirecto las instrucciones definen un registro del procesador que contiene la dirección de memoria donde está el operando. En otras palabras, el registro seleccionado contiene al dirección de memoria donde está almacenado el operando en lugar de tener almacenado el propio operando. Antes de usar una instrucción con modo de direccionamiento indirecto de registro, el programador debe asegurarse de que la dirección de memoria está ya disponible en el registro del procesador. Hacer una referencia al registro es equivalente a especificar una dirección de memoria. La ventaja del modo de registro indirecto es que el campo de

direcciones de la instrucción utiliza menos bits que los que sería necesarios para especificar una dirección de memoria directamente.

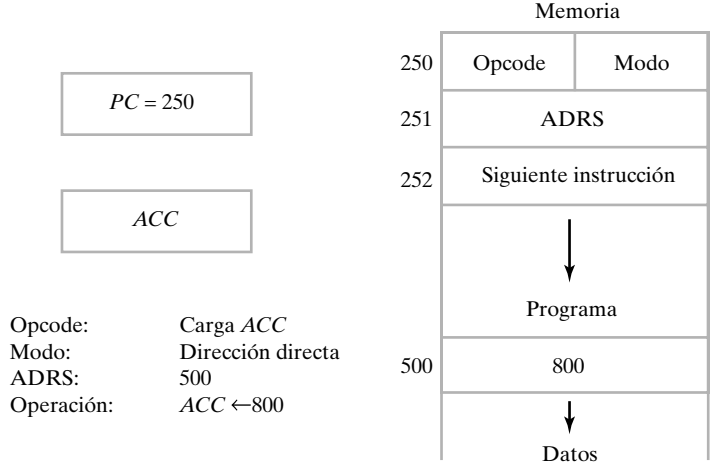
El modo autoincremento o autodecremento es similar al modo de registro indirecto, con la excepción de que el registro incrementa o decrementa después (o antes) de que su valor, dirección, se utilice para acceder a la memoria. Cuando la dirección almacenada en el registro hace referencia a un array de datos de la memoria, es conveniente incrementar el contenido del registro después de cada acceso al array. Esto se puede conseguir mediante una instrucción diferente de incremento de registro. Sin embargo, puesto que es un requisito frecuente, algunos procesadores incorporan un modo de autoincremento que incrementa el contenido del registro que contiene la dirección después de que se ha accedido al dato almacenado en la memoria. En la siguiente instrucción, el modo autoincremento se usa para sumar el valor constante 3 a los elementos de un array direccionado por el registro *R*1:

$$\text{ADD} \qquad (R1) + , 3 \qquad\qquad M[R1] \leftarrow M[R1] + 3,\ R1 \leftarrow R1 + 1$$

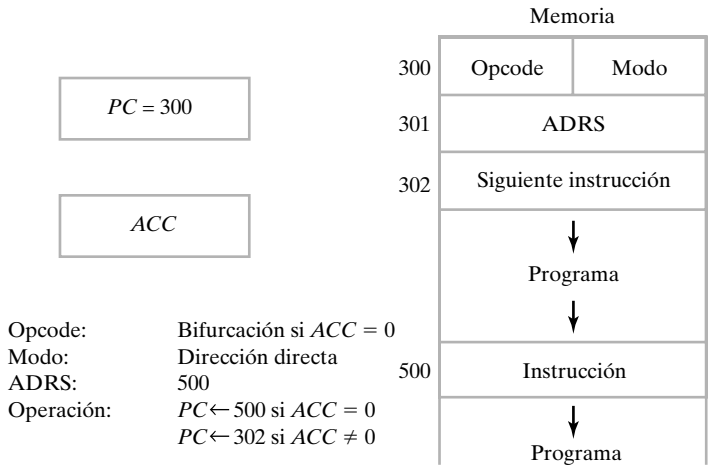
*R*1 se inicializa con la dirección del primer elemento de array. Luego la instrucción ADD se ejecuta repetidamente hasta que se ha efectuado la suma de 3 a todos los elementos del array. La sentencia de transferencia de registros que acompaña a la instrucción muestra la suma de 3 al contenido de la posición de memoria direccionada mediante *R*1 y el incremento de *R*1 para prepararlo para la siguiente ejecución de ADD en el siguiente elemento de array.

Modo de direccionamiento directo

En el modo de direccionamiento directo el campo de direcciones de la instrucción proporciona la dirección de memoria donde está almacenado el operando para ejecutar una instrucción de transferencia de datos o de manipulación de datos. En la Figura 11-4 se muestra un ejemplo de una instrucción de transferencia de datos. La instrucción está compuesta por dos palabras de la memoria. La primera, en la dirección 250, tiene el código de operación para «cargar el *ACC*» y un campo de modo que especifica una dirección directa. La segunda palabra de la instrucción, en la posición 251, contiene el campo de direcciones, simbolizado por ADRS, y que es igual a



□ **FIGURA 11-4**
Ejemplo de direccionamiento directo de una instrucción de transferencia de datos



□ **FIGURA 11-5**
Ejemplo de direccionamiento directo en una instrucción de salto condicional

500. El *PC* guarda la dirección de la instrucción, que se lleva de la memoria utilizando dos accesos a memoria. Simultáneamente con el primer acceso o después de completado éste, el *PC* se incrementa a 251. A continuación se efectúa el segundo acceso para obtener ADRS y, de nuevo, el *PC* se incrementa. La ejecución de la operación da como resultado

$$ACC \leftarrow M[\text{ADRS}]$$

Como ADRS = 500 y $M[500] = 800$, el acumulador recibe el número 800. Después de ejecutarse la instrucción, el *PC* tiene almacenado el número 252, que es la dirección de la siguiente instrucción del programa.

Consideremos ahora una instrucción de salto condicional, como la que se muestra en la Figura 11-5. Si el contenido de *ACC* es igual a 0, el control se bifurca a ADRS; en caso contrario, el programa continúa con la siguiente instrucción de la secuencia. Si $ACC = 0$, la desviación a la dirección 500 se lleva a cabo cargando el valor del campo de direcciones ADRS en el *PC*. El control continúa entonces con la instrucción de la dirección 500. Si $ACC \neq 0$, no se efectúa la bifurcación, y el *PC*, que se ha incrementado dos veces durante el acceso de la instrucción, tiene la dirección 302, que es la dirección de la siguiente instrucción en la secuencia de instrucciones.

Algunas veces, el valor dado en el campo de direcciones es la dirección del operando, pero otras veces es una dirección a partir de la cual se calcula la dirección del operando. Para diferenciar entre los diversos modos de direccionamiento es útil distinguir entre la parte de dirección de la instrucción, dada en el campo de direcciones, y la dirección utilizada para el control cuando se ejecuta una instrucción. Recordemos que hemos llamado a esta última como dirección efectiva.

Modo de direccionamiento indirecto

En el modo de direccionamiento indirecto, el campo de direcciones de la instrucción proporciona la dirección en la que está guardada la dirección efectiva. La unidad de control trae la instrucción de la memoria y utiliza la parte de dirección para acceder de nuevo a la memoria para leer la dirección efectiva. Considere la instrucción «carga en el acumulador» dada en la

Figura 11-4. Si el modo especifica una dirección indirecta, la dirección efectiva está almacenada en *M*[ADRS]. Como ADRS = 500 y *M*[ADRS] = 800, la dirección efectiva es 800. Esto significa que el operando cargado en el *ACC* es el que se encuentra en la dirección 800 de la memoria (no mostrada en la figura).

Modo de direccionamiento relativo

Algunos modos de direccionamiento necesitan que el contenido del campo de la instrucción se sume al contenido de un registro concreto de la CPU para evaluar la dirección efectiva. El registro que más frecuentemente se utiliza es el *PC*. En el modo de direccionamiento, la dirección efectiva se calcula como sigue:

Dirección efectiva = Parte de dirección de la instrucción + Contenido del *PC*

Se considera que la parte de dirección de la instrucción es un número con signo que puede ser positivo o negativo. Cuando este número se suma al contenido del *PC*, el resultado que se genera es una dirección efectiva cuya posición en la memoria es relativa a la dirección de la siguiente instrucción del programa.

Para clarificar esto con un ejemplo, supongamos que el contenido del *PC* contiene el número 250 y la parte de dirección de la instrucción contiene el número 500, como se muestra en la Figura 11-5, con el campo de modo especificando una dirección relativa. Se lee de la memoria la instrucción de la posición 250 en la fase de lectura del ciclo de operación, y el *PC* se incrementa en 1 a 251. Como la instrucción tiene una segunda palabra, la unidad de control lee el campo de dirección de un registro de control, y el *PC* se incrementa a 252. El cálculo de la dirección efectiva para el modo de direccionamiento indirecto es 252 + 500 = 752. El resultado es que el operando asociado a la instrucción se encuentra 500 posiciones más allá, y es relativo a la ubicación de la siguiente instrucción.

El direccionamiento relativo se usa frecuentemente en las instrucciones de salto condicional cuando la dirección de bifurcación está cercana a la palabra de la instrucción. El direccionamiento relativo genera instrucciones más compactas, ya que la dirección relativa puede especificarse con menos bits que los que serían necesarios para designar a una dirección completa de memoria.

Modo de direccionamiento indexado

En el modo de direccionamiento indexado, el contenido de un registro índice se suma a la parte de direcciones de la instrucción para obtener la dirección efectiva. El registro índice puede ser un registro especial de la CPU o simplemente un registro del banco de registros. Ilustraremos el banco de registros considerando un array de datos en la memoria. El campo de direcciones de la instrucción define la dirección de comienzo del array. Cada operando del array se almacena en la memoria en una dirección relativa al comienzo del array. La distancia entre la dirección de comienzo y la dirección del operando es el valor del índice almacenado en el registro. Se puede acceder a cualquier operando del array con la misma instrucción con tal de que el registro índice contenga el valor correcto del índice. El registro índice se puede incrementar para facilitar el acceso a operando consecutivos.

Algunos procesadores dedican un registro de la CPU con la función exclusiva de ser un registro índice. Este registro se direcciona implícitamente cuando se utiliza una instrucción con

modo de direccionamiento indexado. En procesadores con varios registros, cualquiera de éstos se puede usar como registro índice. En tal caso, el registro índice que se va a usar debe especificarse en un campo de registro dentro del formato de la instrucción.

Una variante especial del modo indexado de direccionamiento es el modo de registro base. En este modo, el contenido del registro base se suma a la parte de dirección para obtener la dirección efectiva. Este modo es similar al direccionamiento indexado, excepto que al registro se le llama registro base en lugar de registro índice. La diferencia entre estos dos modos está en la forma en que se usan estos registros más que en la forma en que se calcula la dirección: se supone que un registro índice contiene un número índice que es relativo al campo de direcciones de la instrucción; se supone también que un registro base contiene una dirección base y el campo de instrucciones de la instrucción proporciona un desplazamiento relativo a la dirección base.

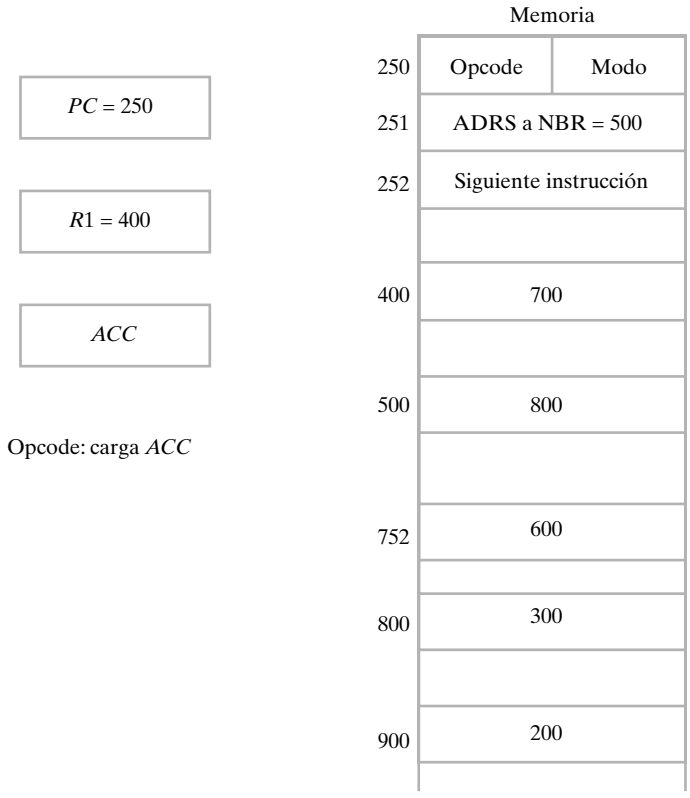
Resumen de modos de direccionamiento

Para mostrar las diferencias entre los diversos modos, investigaremos el efecto del modo de la instrucción que se muestra en la Figura 11-6. La instrucción de la dirección 250 y 251 es «carga el *ACC*», con el campo de direcciones ADRS (o un operando NBR) igual a 500. El *PC* tiene el número 250 para leer esta instrucción. El contenido del registro de procesador *R*1 es 400, y el *ACC* recibe el resultado después de ejecutarse la instrucción. En el modo directo, la dirección efectiva es 500, y el operando que se carga en el *ACC* es 800. En el modo de direccionamiento inmediato, el operando 500 se carga en el *ACC*. En el modo indirecto, la dirección efectiva es 800, y el operando es 300. En el modo relativo, la dirección efectiva 500 + 252 = 752, y el operando es 600. En el modo indexado, el operando está en *R*1, y se carga 400 en el *ACC*. En el modo registro indirecto, la dirección efectiva es el contenido de *R*1 y el operando cargado en *ACC* es 700.

En la Tabla 11-1 se enumeran el valor de la dirección efectiva y el operando cargado en *ACC* para los siete modos de direccionamiento. La tabla muestra también la operación con una sentencia de transferencia de registros y una convención simbólica para cada modo de direccionamiento. LDA es el símbolo para el código de operación de carga al acumulador. En el modo de direccionamiento directo, usamos el símbolo ADRS para la parte de dirección de la instruc-

□ **TABLA 11-1**
Convención simbólica para cada modo de direccionamiento

			Referidos a la Figura 11-6	
Modo de direccionamiento	**Convención simbólica**	**Transferencia de registros**	**Dirección efectiva**	**Contenidos del *ACC***
Directo	LDA ADRS	$ACC \leftarrow M[ADRS]$	500	800
Inmediato	LDA # NBR	$ACC \leftarrow NBR$	251	500
Indirecto	LDA [ADRS]	$ACC \leftarrow M[M[ADRS]]$	800	300
Relativo	LDA $ADRS	$ACC \leftarrow M[ADRS + PC]$	752	600
Indexado	LDA ADRS (R1)	$ACC \leftarrow M[ADRS + R1]$	900	200
Registro	LDA R1	$ACC \leftarrow R1$	—	400
Registro Indirecto	LDA (R1)	$ACC \leftarrow M[R1]$	400	700



□ **FIGURA 11-6**
Ejemplos numéricos para los modos de direccionamiento

ción. El símbolo # precede al operando NBR en el modo inmediato. El símbolo ADRS encerrado en corchetes simboliza una dirección indirecta, la cual es designada por algunos compiladores o ensambladores con el símbolo @. El símbolo $ antes de la dirección hace que la dirección efectiva sea relativa al *PC*. Se reconoce una instrucción con modo indexado mediante el símbolo de un registro colocado entre paréntesis después del símbolo de dirección. El modo registro se indica dando el nombre del registro del procesador a continuación de LDA. En el modo de registro indirecto, el nombre del registro que contiene la dirección efectiva está encerrado entre paréntesis.

11-4 ARQUITECTURAS DE CONJUNTO DE INSTRUCCIONES

Los procesadores tienen un conjunto de instrucciones que permiten llevar a cabo labores de cálculo. Los conjuntos de instrucciones de diferentes procesadores difieren en diversos aspectos unos de otros. Por ejemplo, el código binario asignado al campo del código de operación varía grandemente entre los diferentes procesadores. Asimismo, aunque existe un estándar (*véase* Referencia 7), los mnemónicos de las instrucciones varían de entre los diversos procesadores. Sin embargo, en comparación con estas diferencias menores, hay dos tipos de arquitecturas de conjunto de instrucciones que diferencian marcadamente la relación entre el hardware y el software: procesadores de conjunto de instrucciones complejo (CISC, del inglés *Complex Instruction*

Set Computers) que proporcionan un soporte hardware para operaciones de lenguajes de alto nivel y programas compactos; procesadores de conjunto reducido de instrucciones (RISC, del inglés *Reduced Instruction Set Computers*) que se caracterizan por tener instrucciones sencillas y flexibles que, cuando se combinan, proporcionan un rendimiento más alto y mayor velocidad de ejecución. Estas dos arquitecturas se pueden distinguir considerando las propiedades que caracterizan a sus conjuntos de instrucciones.

Una arquitectura RISC tiene las siguientes propiedades:

1. Los accesos a memora se restingen a las instrucciones de carga y almacenamiento, y las instrucciones de manipulación de datos son de registro a registro.
2. Número limitado de modos de direccionamientos.
3. Los formatos de todas las instrucciones tienen la misma longitud.
4. Las instrucciones realizan operaciones básicas.

Los objetivos de una arquitectura RICS son conseguir un alto rendimiento y una alta velocidad de ejecución. Para conseguir tales objetivos se evitan los accesos a memoria, que típicamente necesitan más tiempo que otras operaciones elementales, excepto las instrucciones de acceso. Una consecuencia de este enfoque es la necesidad de un banco de registros relativamente grande. Debido a que las instrucciones tienen una longitud fija, modos de direccionamiento limitado y operaciones básicas, la unidad de control de un RISC es relativamente simple y, típicamente, está cableada. Además, la organización que subyace es, de forma general, en *pipeline*, como se explica en el Capítulo 12.

Una arquitectura CISC pura tiene las siguientes propiedades:

1. Los accesos a memoria estás directamente disponibles en casi todos los tipo de instrucciones.
2. Mayor número de modos de direccionamientos.
3. Los formatos de las instrucciones son de diferente longitud.
4. Las instrucciones realizan tanto operaciones elementales como complejas.

El objetivo de la arquitectura CISC es ajustarse de forma más cercana a las operaciones empleadas en los lenguajes de programación y proporcionar instrucciones que faciliten realizar programas compactos y conservar memoria. Además, pueden dar lugar a ejecuciones de operaciones eficientes mediante la reducción de los accesos a memoria de las instrucciones. Debido a esta alta accesibilidad de la memoria, los bancos de registros son más pequeños que en una arquitectura RICS. Además, debido a la complejidad de las instrucciones y la diversidad de los formatos de las instrucciones, se suele utilizar el control microprogramado. En la búsqueda para conseguir mayor velocidad, el control microprogramado en los diseños actuales se realiza, probablemente, controlando una ruta de datos en *pipeline*. Las instrucciones CISC se convierten en una secuencia de operaciones de tipo RISC, que se procesan mediante un *pipeline* de tipo RISC, como el que se presenta en el Capítulo 12.

La gama actual de arquitecturas de conjunto de instrucciones está comprendida entre aquellas que son puramente RISC y aquellas que son puramente CISC. No obstante, hay un conjunto básico de operaciones elementales que la mayoría de las computadoras incluyen entre sus instrucciones. En este capítulo, nos centraremos principalmente en las instrucciones elementales que incluyen ambas arquitecturas. La mayoría de las instrucciones elementales de los procesadores se pueden clasificar en tres categorías principales: (1) instrucciones de transferencia de datos, (2) instrucciones de manipulación de datos, y (3) instrucciones de control del programa.

Las instrucciones de transferencia de datos realizan las transferencias de datos desde una posición a otra sin cambiar el contenido de la información binaria. Las instrucciones de manipu-

lación de datos realizan operaciones aritméticas, lógicas y de desplazamiento. Las instrucciones de control del programa proporcionan la capacidad de tomar decisiones y cambian el camino tomado por el programa cuando se ejecutan en el procesador. Aparte del conjunto de instrucciones básicas, un procesador puede tener otras instrucciones que realizan operaciones especiales en aplicaciones concretas.

11–5 INSTRUCCIONES DE TRANSFERENCIA DE DATOS

Las instrucciones de transferencia de datos mueven un dato de un lugar del procesador a otro sin cambiar el dato. Las típicas transferencias son entre memoria y los registros del procesador, entre los registros de procesador y los registros de E/S, y entre los propios registros de procesador.

La Tabla 11-2 proporciona una lista de las ocho transferencias típicas utilizadas en muchos procesadores. A cada instrucción le acompaña su símbolo mnemónico, la abreviatura de lenguaje ensamblador que recomienda el estándar IEEE (Referencia 6). Algunos procesadores, sin embargo, pueden utilizar diferentes mnemónicos para la misma instrucción. La instrucción *Load* se usa para indicar una transferencia desde la memoria a un registro del procesador. La instrucción *Store* indica una transferencia desde un registro del procesador a una posición de memoria. La instrucción *Move* se utiliza en procesadores con varios registros para indicar una transferencia desde un registro del procesador a otro. También se utiliza para la transferencia de datos entre registros y memoria y entre dos posiciones de memoria. La instrucción *Exchange* realiza el intercambio de datos entre dos registros, entre un registro y una posición de memoria, o entre dos posiciones de memoria. Las instrucciones *Push* y *Pop* son para realizar operaciones en la pila y se describen a continuación.

□ **TABLA 11-2**
Instrucciones típicas de transferencia de datos

Nombre	Mnemónico
Load	LD
Store	ST
Move	MOVE
Exchange	XCH
Push	PUSH
Pop	POP
Input	IN
Output	OUT

Instrucciones de manejo de pila

La arquitectura basada en pila, que se presentó anteriormente, posee características que facilitan diversos procesados de los datos y control de las tareas. En algunas calculadoras electrónicas y procesadores se utiliza una pila para evaluar las expresiones aritméticas. Desafortunadamente, debido a los efectos negativos en el rendimiento de las pilas que residen en la memoria principal, la pila de un procesador se utiliza típicamente para almacenar información relacionada con

la llamada a procedimientos, vuelta al programa principal e interrupciones, según se explicó en las Secciones 11-8 y 11-9.

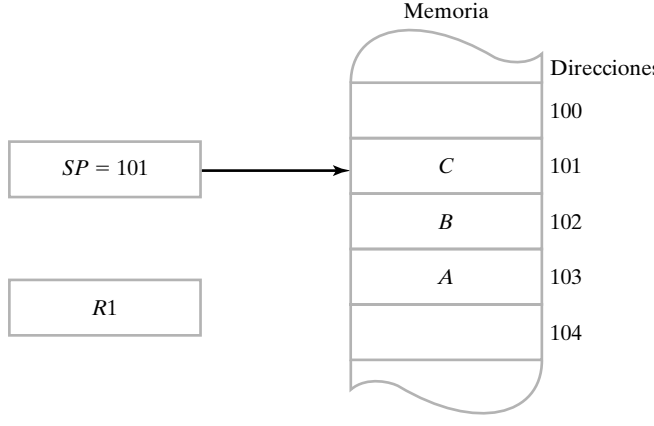
Las instrucciones de pila *push* y *pop* transfieren datos entre la memoria de la pila y un registro de procesador o la memoria. La operación *push* coloca un nuevo objeto en la posición más alta de la pila. La operación *pop* borra un objeto de la pila de forma que el contenido de la pila sube. Sin embargo, nada se mueve realmente en la pila al realizar estas dos operaciones. Más bien, la memoria de la pila es, esencialmente, una porción del espacio de direcciones de memoria accedida por una dirección que siempre se incrementa o decrementa antes o después de acceder a ella. El registro que contiene la dirección para manejar la pila se llama *puntero de pila* (*SP*, del inglés *Stack Pointer*) porque su valor siempre apunta a la posición TOS, que es el dato en lo más alto de la pila. Las operaciones *push* y *pop* realizan el incremento y decremento de puntero de pila.

La Figura 11-7 muestra una parte de una memoria organizada como una pila que crece desde las direcciones más altas hacia las más bajas. El puntero de pila, *SP*, contiene la dirección del dato que está en ese momento en la posición superior de la pila. Hay tres datos almacenados en la pila: *A*, *B* y *C*, en direcciones consecutivas, 103, 102 y 101 respectivamente. El dato *C* está arriba de la pila, por tanto *SP* contiene la dirección 101. Para borrar este dato, la pila «se empuja hacia arriba» leyendo el dato de la dirección 101 e incrementando el *SP*. Ahora el dato *B* está en la posición más alta de la pila ya que *SP* contiene la dirección 192. Para introducir un nuevo dato «se empuja hacia abajo» decrementando primero el *SP* y escribiendo posteriormente el nuevo dato en la posición superior de la pila. Dese cuenta de que el dato *C* se ha sacado de la pila pero, en realidad, no se ha borrado físicamente de ella. Esto no tiene importancia en cuanto a la forma de operar de la pila puesto que, cuando la pila se carga con un nuevo dato, *push*, el que estaba en esa posición queda sobrescrito. Suponemos que los datos en la pila se comunican con un registro de datos *R*1 o con una posición de memoria X.

Un nuevo dato se coloca en la pila con una operación *push* según se indica a continuación:

$$SP \leftarrow SP - 1$$
$$M[SP] \leftarrow R1$$

El puntero de pila se decrementa de forma que apunta a la dirección de la siguiente palabra. Una microoperación de escritura introduce la palabra procedente de *R*1 en la posición más alta



□ **FIGURA 11-7**
Memoria de la pila

de la pila. Véase que *SP* contiene la dirección de la posición más alta de la pila y que *M*[*SP*] designa a la palabra de la memoria especificada por la dirección presente en *SP*. Un dato se elimina de la memoria con una operación *pop* según se indica a continuación:

$$R1 \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

Se lee el dato que está encima de la pila y se lleva a *R*1. El puntero de pila se incrementa para apuntar al siguiente dato de la pila, que se ha convertido en la nueva posición más alta de la pila.

Las dos microoperaciones necesarias para las operaciones *push* y *pop* son: un acceso a la memoria mediante *SP* y una actualización de *SP*. Qué operación se realiza primero y si *SP* se actualiza mediante incremento o decremento, dependerá de la organización de la pila. En la Figura 11-7, la pila crece decrementándose la dirección de memoria. Por el contrario, una pila se puede construir para que crezca mediante incrementos de las direcciones de memoria. En tal caso, *SP* se incrementa en la operación *push* y se decrementa en la operación *pop*. También se puede construir una pila de forma que *SP* apunte a la siguiente posición vacía por encima de la pila. En este caso, la secuencia de operaciones deber intercambiarse.

El puntero de pila se carga con un valor inicial, que debe ser la dirección inferior de la pila asignada en la memoria. A partir de que empiece a funcionar, el *SP* se incrementa y decrementa automáticamente con cada operación *push* o *pop*. La ventaja de una pila en la memoria es que el procesador puede referirse a ella sin tener que especificar una dirección concreta, ya que la dirección está siempre disponible y se puede actualizar automáticamente en el puntero de pila.

Las dos últimas instrucciones de transferencia, entrada y salida, dependen del tipo de E/S, según se describe a continuación.

E/S independiente versus E/S ubicada en memoria

Las instrucciones de E/S transfieren datos entre los registros de procesador y los dispositivos de E/S. Estas instrucciones son similares a las instrucciones de carga y almacenamiento, excepto que las transferencias se realizan con registros externos en lugar de posiciones de memoria. Se supone que el procesador tiene un cierto número de puertos de E/S, con uno o más puertos dedicados a comunicaciones con un dispositivo de E/S concreto. Un puerto es, típicamente, un registro con líneas de entrada y/o salida conectadas a un dispositivo. Un puerto en concreto se selecciona mediante una dirección, de forma similar a la que se selecciona una palabra de la memoria. Las instrucciones de entrada y salida incluyen un campo de direcciones en su formato para especificar el puerto seleccionado para la transferencia de datos.

Las direcciones de los puertos se asignan de dos formas. En los sistemas con E/S independiente, el rango de direcciones asignadas a las memoria y a los puertos de E/S son independientes unas de otras. El procesador tiene instrucciones de entrada y salida diferentes, según se enumeró en la Tabla 11-2, conteniendo un campo de dirección separado que se interpreta por el control y se utiliza para seleccionar un puerto concreto. El direccionamiento de E/S separa la selección de memoria de la E/S, de forma que el rango de direcciones de memoria no se ve afectado por la asignación de las direcciones de los puertos. Por esta razón, a este método también se le llama *configuración separada de E/S*.

Por el contrario, la E/S ubicada en memoria, asigna un subrango de direcciones de memoria para direccionar los puertos de E/S. En este caso no existen direcciones separadas para manejar

las transferencias de entrada y de salida ya que lo puertos de E/S se tratan como posiciones de memoria en un rango de direcciones común. Cada puerto de E/S se ve como una posición de memoria, similar a una palabra de memoria. Los procesadores que adoptan este método no tienen instrucciones específicas para entrada y salida debido a que se usa una misma instrucción para manipular tanto los datos de memoria como los de E/S. Por ejemplo, las instrucciones de carga y almacenamiento utilizadas en la transferencia de memoria también se utilizan para las transferencias de E/S, haciendo que la dirección asociada a la instrucción se asigne a un puerto E/S y no a una palabra de la memoria. La ventaja de este método es la simplicidad que surge de tener el mismo conjunto de instrucciones que dan acceso tanto a la memoria como a la E/S.

11-6 INSTRUCCIONES DE MANIPULACIÓN DE DATOS

Las instrucciones de manipulación de datos realizan operaciones sobre los datos y proporcionan los recursos de cálculo del procesador. En un procesador típico, las instrucciones de manipulación de datos se dividen habitualmente en tres tipos básicos:

1. Instrucciones aritméticas
2. Instrucciones lógicas y de manipulación de bits
3. Instrucciones de desplazamiento

Una lista de las instrucciones de manipulación de datos se parece mucho a la lista de microoperaciones dadas en el Capítulo 10. Sin embargo, una instrucción se procesa típicamente mediante la ejecución de una secuencia de una o más microoperaciones. Una microoperación es una operación elemental ejecutada por el hardware del procesador bajo el control de la unidad de control. Por el contrario, una instrucción puede involucrar varias operaciones básicas como son el acceso a la instrucción, llevar operandos desde los registros adecuados del procesador, y almacenar resultados en las posiciones especificadas.

Instrucciones aritméticas

Las cuatro instrucciones aritméticas básicas son la suma, la sustracción, la multiplicación y la división. La mayoría de los procesadores poseen instrucciones para realizar estas cuatro operaciones. Sin embargo, algunos procesadores pequeños sólo tienen instrucciones de suma y resta; en dichos procesadores, la multiplicación y la división se deben efectuar mediante programas. Estas cuatro operaciones básicas son suficientes para solucionar cualquier problema numérico cuando se utilizan métodos de análisis numérico.

En la Tabla 11-3 se da una lista de las instrucciones aritméticas típicas. La instrucción de incremento suma uno al valor almacenado en un registro o en una palabra de la memoria. Una característica común de la operación de incremento, cuando se ejecuta en una palabra del procesador, es que el número binario con todos sus bits a 1 produce como resultado un número con todos sus bits a 0 cuando se incrementa. La instrucción de decremento resta uno al valor almacenado en un registro del procesador o en una posición de memoria. Cuando se decrementa un número con todos sus bits a 0 se produce como resultado un número con todos sus bits a 1.

Las instrucciones de suma, sustracción, multiplicación y división pueden estar disponibles para diferentes tipos de datos. Se supone que el tipo de dato que va a estar en un registro del procesador durante la ejecución de estas operaciones aritméticas se incluye en la definición del código de operación. Una operación aritmética puede especificar datos enteros con signo o sin

□ **TABLA 11-3**
Instrucciones aritméticas típicas

Nombre	Mnemónico
Incremento	INC
Decremento	DEC
Suma	ADD
Resta	SUB
Multiplicación	MUL
División	DIV
Suma con acarreo	ADDC
Resta con acarreo	SUBB
Resta inversa	SUBR
Negación	NEG

signo, números decimales o binarios o datos en punto flotante. Las operaciones con enteros en binario se presentaron en los Capítulos 1 y 5. La representación en punto flotante se utiliza en cálculos científicos y se presenta en la siguiente sección.

El número de bits de cualquier registro es finito, por tanto, el resultado de las operaciones aritméticas tiene una precisión finita. La mayoría de los procesadores tiene instrucciones especiales que facilitan trabajar con doble precisión aritmética. Se usa un flip-flop para almacenar la información de acarreo. La instrucción «suma con acarreo» realiza la operación de dos operandos más el valor del acarreo del cálculo anterior. De forma similar, la instrucción de resta con acarreo (*borrow*) sustrae dos operandos y el acarreo que se generó en la operación previa. La instrucción de resta inversa invierte el orden de los operando, realizando la operación $B - A$ en lugar de $A - B$. La instrucción negación realiza el complemento a 2 de un número con signo, que es equivalente a multiplicar el número por -1.

Instrucciones lógicas y de manipulación de bits

Las instrucciones lógicas realizan operaciones en las palabras almacenadas en registros o en memoria. Son útiles para la manipulación individual de bits o de grupos de bits que representan información codificada en binario. Las instrucciones tratan cada bit del operando por separado como si fuesen variables booleanas. Mediante la aplicación adecuada de las instrucciones lógicas es posible cambiar los valores de un bit, poner a cero un grupo de bits o introducir un valor nuevo de un bit en los operandos almacenados en los registros o en la memoria.

En la Tabla 11-4 se enumeran algunas instrucciones lógicas típicas. La instrucción *Clear* hace que el operando especificado sea reemplazado por ceros. La instrucción *Set* provoca que el operando sea reemplazado por unos. La instrucción Complemento invierte el valor de todos los bits del operando. Las instrucciones AND, OR y XOR realizan la operación lógica correspondiente en los bits del operando individualizadamente. Aunque las instrucciones lógicas realizan operaciones booleanas, cuando éstas se efectúan en palabras, se suelen considerar como operaciones de manipulación de bits. Hay tres operaciones posibles para manipular bits: poner un bit seleccionado a 0, ponerlo a 1 o complementarlo. Las tres operaciones lógicas se emplean habitualmente para realizar estas operaciones. La instrucción AND se usa para poner un bit o grupo

□ **TABLA 11-4**
Instrucciones lógicas y de manipulación de bits típicas

Nombre	Mnemónico
Puesta a 0 (*Clear*)	CLR
Puesta a 1 (*Set*)	SET
Complemento	NOT
AND	AND
OR	OR
OR Exclusiva	XOR
Poner a 0 el acarreo (*Clear carry*)	CLRC
Poner a 1 el acarreo (*Set carry*)	SETC
Complemento del acarreo	COMC

de bits de un operando a 0. Para cualquier variable booleana X, la expresión $X \cdot 0 = 0$ indica que una variable binaria multiplicada (AND) por 0 produce un 0; y de forma similar, la expresión $X \cdot 1 = X$ indica que una variable no cambia al ser multiplicada por 1. Por tanto, la instrucción AND se usa para poner los bits de un operando a 0 de forma selectiva mediante la operación AND con una palabra que contenga 0 en las posiciones de los bits que se quieren poner a 0, y el resto permanecen sin cambiar. A la operación AND también se le llama máscara porque, insertando ceros, enmascara una parte del operando. A la instrucción AND también se le denomina instrucción de bits *Clear*.

La instrucción OR se usa para poner un bit o un grupo de bits de un operando a 1. Para cualquier variable X, la expresión $X + 1 = 1$ indica que una variable a la que se le aplica la suma lógica (OR) de 1 produce un 1; de forma similar, la expresión $X + 0 = X$, indica que la variable permanece sin cambiar al aplicar la suma lógica con 0. Por tanto, la instrucción OR se puede usar para poner selectivamente a 1 los bits de un operando mediante la operación OR con una palabra que contenga unos en la posición de los bits que se quieren poner a 1. A veces, a la instrucción OR se le llama instrucción de bits *Set*.

La instrucción XOR se utiliza para complementar selectivamente los bits de un operando. Esto se debe a la relación booleana $X \oplus 1 = \bar{X}$ y $X \oplus 0 = X$. Una variable booleana se complementa cuando se realiza la operación XOR con 1, pero no cambia cuando se realiza la operación XOR con 0. A veces, a la instrucción XOR se le llama instrucción de complemento de bit.

En la Tabla 11.4 se incluyen otras instrucciones de manipulación de bits, puesta a 0 (*Clear*), puesta a 1 (*Set*) o complemento del bit de acarreo. Se pueden realizar estas mismas operaciones en los bits de status de manera similar.

Instrucciones de desplazamiento

Existe una serie de instrucciones para desplazar el contenido de un operando de diversas formas. Los desplazamientos son operaciones en las que los bits del operando se mueven a la izquierda o a la derecha. El bit entrante en el desplazamiento de la palabra determina el tipo de desplazamiento. Aquí hemos añadido otras posibilidades aparte de utilizar un 0 entrante, según se hizo en los desplazamientos *sl* y *sr* del Capítulo 10. Las instrucciones de desplazamiento pueden especificar desplazamientos lógicos, aritméticos u operaciones de rotación.

En la Tabla 11-5 se enumeran los cuatro tipos de instrucciones de desplazamiento. El desplazamiento lógico introduce 0 en la posición del bit entrante después del desplazamiento. Los desplazamientos aritméticos se ajustan a las reglas para el desplazamiento de números con signo en complemento a 2. El desplazamiento aritmético a la derecha conserva el bit de signo en la posición más a la izquierda. El valor del bit de signo se desplaza hacia la derecha junto con el resto del número pero manteniendo el bit de signo sin cambiar. La instrucción de desplazamiento aritmético a la izquierda introduce 0 como bit entrante en la posición más a la derecha y es idéntica a la instrucción de desplazamiento lógico hacia la izquierda. Sin embargo, las dos instrucciones se pueden distinguir en que en el desplazamiento aritmético a la izquierda se puede poner a 1 el bit de status *overflow*, *V*, mientras que el desplazamiento lógico a la izquierda no afecta a *V*.

□ **TABLA 11-5**
Instrucciones de desplazamiento típicas

Nombre	Mnemónico
Desplazamiento lógico a la derecha	SHR
Desplazamiento lógico a la izquierda	SHL
Desplazamiento aritmético a la derecha	SHRA
Desplazamiento aritmético a la izquierda	SHLA
Rotación a la derecha	ROR
Rotación a la izquierda	ROL
Rotación a la derecha con acarreo	RORC
Rotación a la izquierda con acarreo	ROLC

Las instrucciones de rotación producen un desplazamiento circular: los valores que salen al desplazarse por el bit de salida de la palabra no se pierden, como en el desplazamiento lógico, sino que se introducen por el bit entrante. Las instrucciones de rotación con acarreo tratan al bit de acarreo como una extensión del registro cuya palabra esta siendo rotada. De esta forma, una rotación a la izquierda con acarreo transfiere el bit de acarreo al bit de entrada a la posición más a la derecha del registro, transfiere el bit saliente desde el bit más a la izquierda del registro al acarreo y desplaza el contenido completo del registro a la izquierda. Algunos procesadores tienen un formato con varios campos para la instrucción de desplazamiento. Un campo contiene el código de operación y el resto especifica el tipo de desplazamiento y el número de posiciones que el operando se va a desplazar. Una instrucción de desplazamiento puede incluir los siguientes cinco campos:

OP REG TYPE RL COUNT

OP es el campo del código de operación para especificar un desplazamiento y REG tiene una dirección que especifica la localización del operando. TYPE es un campo de dos bits que especifica uno de los cuatro tipos de desplazamiento (lógico, aritmético, rotación y rotación con acarreo), y RL es un campo de 1 bit que indica si el desplazamiento es a la derecha o a la izquierda. COUNT es un campo de k bits que indica el número de posiciones a desplazar, hasta un máximo de $2^k - 1$ posiciones. Con dicho formato, es posible especificar el tipo de desplazamiento, su dirección y el número de posiciones que se ha de desplazar el operando, todo ello en una instrucción.

11-7 CÁLCULOS EN PUNTO FLOTANTE

En muchos cálculos científicos, el rango de los números que se utilizan es muy grande. En un procesador, la forma de expresar tales números es en notación en punto flotante. Los números en punto flotante tienen dos partes: una contiene el signo del número y una *fracción* o *mantisa*, y la otra parte indica la posición del punto de la base y se llama *exponente*. Por ejemplo, el número decimal $+6132.789$ se representa en notación en punto flotante como:

Mantisa	Exponente
$+.6132789$	$+04$

El valor del exponente indica que la posición actual del punto decimal está cuatro posiciones a la derecha del punto decimal indicado en la mantisa. Esta representación es equivalente a la notación científica $+1.6132789 \times 10^{+4}$. Los números decimales en punto flotante se interpretan para representar un número de la forma:

$$F \times 10^{E}$$

donde F es la mantisa y E el exponente. Sólo la mantisa y el exponente están físicamente representado en los registros del procesador. La base 10 y el punto decimal de la mantisa no se muestran explícitamente. Un número binario en punto flotante se representa de forma similar, excepto que se usa la base 2 para el exponente. Por ejemplo, el número binario $+1001.11$ se representa con 8 bits para la mantisa y 6 bits para el exponente como

Mantisa	Exponente
01001110	000100

La mantisa tiene un 0 en la posición más a la izquierda que indica un $+$. El punto binario de la mantisa sigue al bit de signo pero no se muestra en el registro. El exponente tiene el número binario equivalente a $+4$. El número en punto flotante es equivalente a:

$$F \times 2^{E} = +(0.1001110)_2 \times 2^{+4}$$

Se dice que un número en punto flotante está normalizado si el digito más significativo de la mantisa es distinto de cero. Por ejemplo, la mantisa decimal 0.350 está normalizada pero 0.0035 no. Los números normalizados proporcionan la máxima precisión de los números en punto flotante. El cero no puede ser normalizado puesto que no tiene dígitos distintos de cero y se representa normalmente en punto flotante con todos sus bits a cero, tanto en la mantisa como en el exponente.

La representación en punto flotante incrementa el rango de los números que se pueden ajustar a un determinado registro. Suponga un procesador con registros de 48 bits. Puesto que se tiene que reservar un bit para el signo, el rango de los enteros con signo será $\pm(2^{47} - 1)$, que es aproximadamente $\pm 10^{14}$. Los 48 bits se pueden usar para representar un número en punto flotante, con un bit de signo, 35 bits para la mantisa y 12 bits para el exponente. Los números positivos y negativos más grandes que se pueden representar así son:

$$\pm(1 - 2^{-35}) \times 2^{+2047}$$

Este número se deriva de una mantisa que contiene 35 unos y un exponente con un bit de signo y 11 unos. El exponente más grande es $2^{11} - 1$, o 2047. El número decimal más grande que se puede representar así es equivalente a 10^{615} aproximadamente. Aunque se represente un rango mucho mayor, sólo hay 48 bits en la representación. Como consecuencia, se pueden representar la misma cantidad de números. Por ello, el rango se ajusta según la precisión de los números, que se reduce de 48 bits a 35 bits.

Operaciones aritméticas

Las operaciones aritméticas con número en punto flotante son más complicadas que con números enteros, y su ejecución lleva más tiempo y requiere un hardware más complejo. La suma y la resta de dos números necesitan que se alineen los puntos de la base ya que los exponentes deben ser iguales antes de sumar o de restar las fracciones. Considere la suma de los siguientes números en punto flotante:

$$\begin{array}{r} .5372400 \times 10^{2} \\ +.1580000 \times 10^{-1} \end{array}$$

Es necesario que los dos exponentes sean iguales antes de que se puedan sumar las fracciones. Podemos desplazar el primer número tres posiciones a la izquierda o desplazar el segundo número tres posiciones a la derecha. Si las fracciones están almacenadas en registros, el desplazamiento a la derecha puede provocar la pérdida de los dígitos menos significativos. El segundo método es preferible porque solamente reduce la precisión, mientras que el primer método puede causar un error. El procedimiento habitual de alineamiento es desplazar la mantisa con el exponente menor a la derecha un número de posiciones igual a la diferencia entre los exponentes. Después de esto, las fracciones se pueden sumar:

$$\begin{array}{r} .5372400 \times 10^{2} \\ +.0001580 \times 10^{2} \\ \hline .5373980 \times 10^{2} \end{array}$$

Cuando se suman dos fracciones, la suma puede contener un dígito de *overflow*. Un *overflow* se puede corregir desplazando la suma una vez a la derecha e incrementando el exponente. Cuando se restan dos números, el resultado puede contener ceros en los dígitos más significativos, como se muestra en el siguiente ejemplo:

$$\begin{array}{r} .56780 \times 10^{5} \\ -.56430 \times 10^{5} \\ \hline .00350 \times 10^{5} \end{array}$$

Un número en punto flotante que tiene un 0 en la posición más significativa de la mantisa no es un número normalizado. Para normalizar el número es necesario desplazar la mantisa a la izquierda y decrementar el exponente hasta que aparezca un dígito distinto de cero en la primera posición. En el ejemplo anterior, es necesario desplazar el resultado dos veces hacia la izquierda hasta obtener $.35000 \times 10^{3}$. En la mayoría de los procesadores, el procedimiento de normalización se produce después de cada operación para asegurar que todos los resultados están de forma normalizada.

La multiplicación y división en punto flotante no necesita un alineamiento de las fracciones. La multiplicación se puede realizar multiplicando las dos fracciones y sumando los exponentes. La división se puede llevar a cabo dividiendo las fracciones y restando los exponentes. En los ejemplos mostrados, utilizamos números decimales para mostrar las operaciones aritméticas con números en punto flotante. El mismo procedimiento se aplica a los números binarios, excepto que la base del exponente es 2 en lugar de 10.

Exponente sesgado

La parte del signo y la mantisa de un número en punto flotante se representan habitualmente en código signo más magnitud. La representación de exponentes empleada en la mayoría de los procesadores se conoce como exponente sesgado (del término inglés *biased exponent*). El sesgo es un valor fijo en exceso que se suma al exponente de forma que, internamente, todos los exponentes pasan a ser positivos. Como consecuencia, el signo del exponente se elimina y no forma una entidad separada.

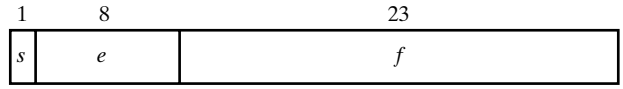
Consideremos, por ejemplo, que el rango de los exponentes decimales está entre -99 y $+99$. Esto se representa mediante dos dígitos más el signo. Si utilizamos como sesgo el valor 99, entonces el exponente sesgado e será igual a $e = E + 99$, donde E es el exponente actual. Para $E = -99$, tendremos que $e = -99 + 99 = 0$; y para $E = +99$, tendremos que $e = 99 + 99 = 198$. De esta forma, el exponente sesgado se representa en un registro como un número positivo en el rango de 000 a 198. Los exponentes sesgados positivos tienen un rango entre 099 y 198. La resta del sesgo, 99, da valores positivos entre 0 y $+99$. Los exponentes segados negativos tendrán un rango entre 098 y 000. La resta de 99 da los valores negativos comprendidos entre -1 y -99.

La ventaja de los exponentes sesgados es que los números en punto flotante que resultan contienen sólo exponentes positivos. Es, por tanto, más sencillo comparar la magnitud relativa entre dos números sin tener en cuenta los signos de sus exponentes. Otra ventaja es que el exponente más negativo se convierte a exponente sesgado con todos los dígitos a 0. La representación de cero es un cero en la mantisa y un cero en el exponente sesgado, que es el exponente más pequeño.

Formato estándar de los operandos

Las instrucciones aritméticas que realizan operaciones con datos en punto flotante utilizan habitualmente el sufijo F. De esta forma, ADDF es una instrucción de suma con números en punto flotante. Hay dos formatos estándar para la representación de operandos en punto flotante: datos en precisión simple, consistente en 32 bits, y datos en doble precisión, consistente en 64 bits. Cuando están disponibles los dos tipos de datos, el mnemónico de las instrucciones de precisión simple utilizan el sufijo FS, y los de doble precisión usan FL (del término inglés *floating-point long*).

El formato estándar del IEEE (*véase* Referencia 7) para precisión simple en operando en punto flotante se muestra en la Figura 11-8. Está compuesto por 32 bits. El bit de signo s indica el signo de la mantisa. El exponente sesgado, e, tiene 8 bits y utiliza como sesgo el número 127. La mantisa, f, está formada por 23 bits. Se considera que el punto binario está inmediatamente a la izquierda del bit más significativo del campo f. Además, implícitamente se inserta un bit a la izquierda del punto binario, el cual, expande el número a 24 bits, representando un número de valores entre 1.0_2 a $1.11...1_2$. A la componente del número binario en punto flotante, que consiste



□ **FIGURA 11-8**
Formato IEEE para los operandos en punto flotante

en un bit primero a la izquierda del punto binario implícito, junto con la mantisa en el campo, se le llama *parte significativa*. A continuación tenemos algunos ejemplos de valores del campo y su correspondiente parte significativa:

Campo *f*	Parte significativa	Equivalente decimal
100 . . . 0	1.100 . . . 0	1.50
010 . . . 0	1.010 . . . 0	1.25
000 . . . 0	1.000 . . . 0*	1.00*

* Suponiendo que el exponente es distinto de 00 . . . 0.

Aunque el campo f no esté normalizado, la parte significativa está siempre normalizada porque no tiene un bit distinto de cero en el bit más significativo. Puesto que los números normalizados deben tener el bit más significativo distinto de cero, este bit no se incluye en el formato pero se debe introducir por el hardware durante los cálculos aritméticos. El componente utiliza 127 como valor de sesgo para los números normalizados. El rango válido para los exponentes va desde -126 (representado como 00000001) hasta $+127$ (representado como) 11111110. El valor máximo (11111111) y mínimo (00000000) del campo *e* se reservan para indicar condiciones excepcionales. La Tabla 11-6 muestra los valores de sesgo y actuales de algunos exponentes.

±□ **TABLA 11-6**
Evaluación de exponentes sesgados

Exponente *E* en decimal	Exponente sesgado $e = E + 127$	
	Decimal	**Binario**
−126	−126 + 127 = 1	00000001
−001	−001 + 127 = 126	01111110
000	000 + 127 = 127	01111111
+001	001 + 127 = 128	10000000
+126	126 + 127 = 253	11111101
+127	127 + 127 = 254	11111110

Los números normalizados son números que se pueden expresar como operandos en punto flotante en los que en el campo *e* no están todos los bits ni a 1 ni a 0. El valor del número se extrae de los tres campos del formato de la Figura 11-8 utilizando la fórmula

$$(-1)^s 2^{e-127} \times (1.f)$$

El número más positivo normalizado que se puede obtener tiene un 0 en el bit de signo para indicar el signo positivo, un exponente sesgado igual a 254 y un campo f con 23 unos. Esto da un exponente $E = 254 - 127 = 127$. La parte significativa es igual a $1 + 1 - 2^{-23} = 2 - 2^{-23}$. El número positivo máximo que se puede utilizar es:

$$+2^{-127} \times (222^{-23})$$

El número normalizado positivo más pequeño tiene un exponente sesgado igual a 00000001 y una mantisa con todos sus bits a 0. El exponente es $E = 1 - 127 = -126$, y la parte significativa es igual a 1.0. El número positivo más pequeño que se puede usar es $+2^{-126}$. Los números negativos correspondientes a los anteriores son los mismos, excepto que el bit de signo es negativo. Como se mencionó anteriormente, los exponentes con todos sus bits a 0 o a 1 (en decimal 255) se reservan para las siguientes condiciones especiales:

1. Cuando $e = 255$ y $f = 0$, el número representa más o menos infinito. El signo lo determina el bit de signo s.
2. Cuando $e = 255$ y $f \neq 0$, la representación no se considera como un número o NaN (del inglés Not a Number), ignorando el bit de signo. NaN se utiliza para indicar operaciones no válidas, como puede ser la multiplicación de cero por infinito.
3. Cuando $e = 0$ y $f = 0$, el número es un más o menos cero.
4. Cuando $e = 0$ y $f \neq 0$, se dice que el número no está normalizado. Este es el nombre que se le da a los números cuya magnitud es menor que el valor mínimo que se puede representar con formato normalizado.

11-8 INSTRUCCIONES DE CONTROL DE PROGRAMA

Las instrucciones de un programa se almacenan en posiciones de memoria consecutivas. Cuando se procesan por la unidad de control, las instrucciones se leen de las posiciones consecutivas de la memoria y se ejecutan una por una. Cada vez que se extrae una instrucción de la memoria, el *PC* se incrementa de forma que contiene la siguiente dirección de la secuencia. Por el contrario, una instrucción de control de programa, cuando se ejecuta, puede cambiar el valor de la dirección del *PC* y alterar el flujo de control. El cambio en el *PC*, como resultado de la ejecución de una instrucción de control de programa, provoca una ruptura en la secuencia de ejecución de las instrucciones. Esta es una característica importante en los procesadores digitales ya que proporciona un control sobre el flujo de ejecución del programa y la posibilidad de bifurcarse hacia diferentes segmentos del programa, dependiendo de los cálculos previos.

En la Tabla 11-7 se enumeran algunas instrucciones de control de programa típicas. La bifurcación, o ramificación, (del término ingles *branch*) y el salto (del término ingles *jump*) se utilizan, con frecuencia, indistintamente, significando la misma cosa, aunque a veces, se usan con modos de direccionamiento diferentes. Por ejemplo, el salto puede usar direccionamiento directo o indirecto, mientras que la bifurcación utiliza direccionamiento relativo. Estos saltos y bifurcaciones son, habitualmente, instrucciones con una dirección. Cuando se ejecutan, la bifurcación realiza la transferencia de la dirección efectiva al *PC*. Como el *PC* contiene la dirección de la instrucción que se va a ejecutar posteriormente, la siguiente instrucción se extraerá de la posición especificada por la dirección efectiva.

Las instrucciones de salto y ramificación pueden ser condicionales o incondicionales. Una instrucción de ramificación incondicional realiza una bifurcación a una dirección efectiva sin ninguna condición. La instrucción de bifurcación especifica una condición que se debe cumplir

□ **TABLA 11-7**
Instrucciones de control de programa típicas

Nombre	Mnemónico
Bifurcación	BR
Salto	JMP
Salto implícito (*Skip*)	SKP
Llamada a subrutina	CALL
Retorno de subrutina	RET
Comparación (con substracción)	CMP
Test (mediante AND)	TEST

para que la ramificación se produzca, como puede ser que el valor de un registro concreto sea negativo. Si se cumple la condición, el *PC* se carga con la dirección efectiva y la siguiente instrucción a ejecutar se toma de esta dirección. Si la condición no se cumple, el contenido del *PC* no cambia, y la siguiente instrucción se toma de la siguiente posición de la secuencia.

La instrucción de salto implícito (en inglés *skip*) no necesita campo de direcciones. Una instrucción de salto implícito condicional saltará a la siguiente instrucción de la secuencia si se cumple una determinada condición, quedándose ésta sin ejecutar. Esto se realiza mediante el incremento del *PC* durante la fase de ejecución de la instrucción, además de incrementarla durante la fase de extracción. Si no se cumple, el control prosigue con la siguiente instrucción de la secuencia, donde el programador puede insertar una instrucción de ramificación incondicional. De esta forma, una instrucción de salto implícito seguida de una instrucción de bifurcación provoca una ramificación si la condición no se cumple. Como este salto involucra la ejecución de dos instrucciones, resulta más lento y usa más instrucciones de la memoria.

Las instrucciones de llamada y de retorno de subrutina se utilizan para usar subrutinas. Su funcionamiento se presenta más tarde en esta sección.

La instrucción de comparación realiza una comparación mediante una resta pero sin almacenar el resultado. En cambio, la comparación realiza una bifurcación condicional, cambia el contenido de un registro o pone a 0 o a 1 los bits de status. De forma similar, la instrucción de test efectúa una operación AND entre dos operando sin almacenar el resultado y ejecuta una de las acciones enumeradas en la instrucción de comparación.

Según las tres acciones posibles, las instrucciones de comparación y de test se pueden clasificar en tres tipos, dependiendo de la forma en la que se manejen las decisiones condicionales. El primer tipo ejecuta la decisión en una sola instrucción. Por ejemplo, se puede comparar el contenido de dos registros y se realiza una bifurcación o un salto si sus contenidos son iguales. Al estar involucradas dos direcciones de registro y una dirección de memoria, dicha instrucción necesita tres direcciones. El segundo tipo de instrucción de comparación y de test también utiliza tres direcciones pero esta vez sólo de registros. Considerando el mismo ejemplo, si el contenido de los dos primeros registros es igual, se coloca un 1 en el tercer registro. Si el contenido no es igual, entonces se coloca un 0 en el tercer registro. Estas instrucciones evitan el uso de los bits de status. En el primer caso, tales bits no se necesitan, y en el segundo caso, se usa un registro para simular la presencia de un bit de status. El tercer tipo de comparación y test, con la estructura más compleja, tiene operaciones de comparación y test que ponen a 1 o a 0 los bits de status. Las instrucciones de bifurcación o de salto se utilizan posteriormente para cambiar condicionalmente la secuencia del programa. Este tercer tipo de instrucción de comparación y test centrará la discusión de la siguiente subsección.

Instrucciones de bifurcación condicional

Una instrucción de bifurcación condicional es una instrucción de ramificación que puede o no provocar una transferencia de control, dependiendo de los valores de los bits almacenados en el *PSR*. Cada instrucción de bifurcación condicional comprueba una combinación diferente de los bits de status dependiendo de la condición. Si la condición es cierta, el control se transfiere a la dirección efectiva. Si la condición es falsa, el programa continúa con la siguiente instrucción. En la Tabla 11-8 se da una lista de las instrucciones condicionales que dependen directamente de los bits de *PSR*. En la mayoría de los casos, el mnemónico se construye con la letra B (de *branch*) y una letra que nombra al bit de status involucrado. Se añade la letra N (de *not*) si el bit de status involucrado se comprueba para la condición 0. De esta forma, BC efectúa una ramificación si el acarreo es igual a 1, y BNC hace una ramificación si el bit de acarreo es igual a 0.

□ **TABLA 11-8**
Instrucciones de fifurcación relacionadas con los bits de status del PSR

Condición de bifurcación	Mnemónico	Condición de test
Bifurcación si es cero	BZ	$Z = 1$
Bifurcación si no es cero	BNZ	$Z = 0$
Bifurcación si hay acarreo	BC	$C = 1$
Bifurcación si no hay acarreo	BNC	$C = 0$
Bifurcación si negativo	BN	$N = 1$
Bifurcación si positivo	BNN	$N = 0$
Bifurcación si hay *overflow*	BV	$V = 1$
Bifurcación si no hay *overflow*	BNV	$V = 0$

El bit de status cero, *Z*, se usa para comprobar si el resultado de una operación de la ALU es igual a cero. El bit de acarreo se utiliza para comprobar el acarreo después de una suma o una resta de dos operandos en la ALU. También se utiliza junto con las instrucciones de desplazamiento para comprobar el bit saliente. El bit de signo, *N*, refleja el estado del bit más a la izquierda de la salida de la ALU. $N = 0$ indica signo positivo y $N = 1$ signo negativo. Estas instrucciones se pueden utilizar para comprobar el valor del bit más a la izquierda, tanto si representa un signo o no. El bit de *overflow*, *V*, se usa junto con operaciones aritméticas con números con signo.

Como se explicó anteriormente, la instrucción de comparación realiza la resta de dos operandos, supongamos, $A - B$. El resultado de la operación no se transfiere a ningún registro, aunque los bits de status resultan afectados. Los bits de status proporcionan información sobre la diferencia entre *A* y *B*. Algunos procesadores tienen instrucciones de bifurcación que se pueden aplicar después de la ejecución de una instrucción de comparación. Las condiciones concretas que se comprueban dependen de si los números se consideran con signo o sin signo.

La diferencia entre dos números binarios sin signo *A* y *B* se puede determinar mediante la operación $A - B$ y comprobando los bits de status *C* y *Z*. La mayoría de procesadores tienen en cuenta al bit de status *C* como bit de acarreo después de una suma o una resta. Se produce un acarreo en la substracción cuando $A < B$ porque la posición más significativa debe prestar un bit para completar la resta. Ese acarreo no ocurre si $A \geqslant B$, ya que $A - B$ es positivo. La condición de acarreo en la substracción es la contraria si la resta se hace con el complemento a 2 de *B*. Los procesadores que utilizan el bit de status *C* como acarreo después de una resta comple-

mentan la salida de acarreo después de sumar el complemento a 2 del sustraendo y llama a este bit «bit de *borrow*». La técnica se aplica a todas las instrucciones que usan la resta dentro de la unidad funcional, no sólo para la instrucción de resta. Por ejemplo, se aplica para la instrucción de comparación.

Las instrucciones de bifurcación condicional con números sin signo se enumeran en la Tabla 11-9. Se considera que la instrucción anterior ha actualizad los bits de status C y Z después de la operación $A - B$ u otra instrucción similar. Las palabras «mayor», «menor» e «igual» se usan para indicar la diferencia entre dos números sin signo. Los dos números son iguales si $A = B$. Esto se determina a partir del bit de status Z, que es igual a 1 ya que $A - B = 0$. A es menor que B y el acarreo $C = 1$ cuando $A < B$. Para que A sea menor o igual que B, ($A \leqslant B$), debemos tener $C = 1$ o $Z = 1$. La relación $A > B$ es la inversa de $A \leqslant B$ y se detecta complementando la condición de los bits de status. De forma similar, $A \geqslant B$, es la inversa de $A < B$, y $A \neq B$ es la inversa de $A = B$.

□ **TABLA 11-9**
Instrucciones de bifurcación condicional para números sin signo

Condición de bifurcación	Mnemónico	Condición	Bits de status
Bifurcación si es mayor	BH	$A > B$	$C + Z = 0$
Bifurcación si es mayor o igual	BHE	$A \geqslant B$	$C = 0$
Bifurcación si es menor	BL	$A < B$	$C = 1$
Bifurcación si es menor o igual	BLE	$A \leqslant B$	$C + Z = 1$
Bifurcación si es igual	BE	$A = B$	$Z = 1$
Bifurcación si no es igual	BNE	$A \neq B$	$Z = 0$

Las instrucciones de salto condicional para números con signo se enumeran en la Tabla 11-10. De nuevo se supone que una instrucción anterior ha actualizado los bits de status N, V y Z después de la operación $A - B$, Las palabras «más grande», «menos grande» e «igual» se usan para indicar la diferencia entre dos números con signo. Si $N = 0$, el signo de la diferencia es positiva y A debe ser más grande o igual que B, haciendo $V = 0$, e indicando que no ha ocurrido *overflow*. Un *overflow* produce un cambio de signo, según se estudió en la Sección 5-4. Esto significa que si $N = 1$ y $V = 1$, hubo un cambio de signo y que el resultado debió haber sido positivo, lo cual hace A más grande o igual que B. Además, la condición $A \geqslant B$ es cierta si N y V son iguales a 0 o ambos son iguales a 1. Esto es el complemento de la operación OR exclusiva.

Para que A sea mayor que B pero no igual ($A > B$), el resultado debe ser positivo y distinto de cero. Como un resultado igual a cero tiene signo positivo, debemos asegurarnos que el bit Z

□ **TABLA 11-10**
Instrucciones de bifurcación condicional para números con signo

Condición de bifurcación	Mnemónico	Condición	Bits de status
Bifurcación si es mayor	BG	$A > B$	$(N \oplus V) + Z = 0$
Bifurcación si es mayor o igual	BGE	$A \geqslant B$	$(N \oplus V) = 0$
Bifurcación si es menor	BL	$A < B$	$(N \oplus V) = 1$
Bifurcación si es menor o igual	BLE	$A \leqslant B$	$(N \oplus V) + Z = 1$

es 0 para excluir la posibilidad de que $A = B$. Véase que la condición $(N \oplus V) + Z = 0$ significa que la operación OR exclusiva y Z deben ser iguales a 0. Las otras condiciones de la tabla se pueden derivar de manera similar. Las condiciones BE (bifurcación si es igual) y BNE (bifurcación si no es igual) dadas para los números sin signo se aplican también a los números con signo y se pueden determinar a partir de $Z = 1$ y $Z = 0$, respectivamente.

Instrucciones de llamada y retorno de subrutinas

Una *subrutina* es una secuencia autocontenida de instrucciones que realizan una tarea de cálculo concreta. Durante la ejecución de un programa, se puede llamar a una subrutina varias veces en diversos puntos de un programa. Cada vez que se llama a la subrutina se realiza una bifurcación al comienzo de ésta para empezar la ejecución de su conjunto de instrucciones. Después de ejecutar la subrutina se realiza una nueva ramificación para volver al programa principal.

La instrucción que transfiere el control a la subrutina se conoce con diversos nombres: llamadas subrutinas, salto a subrutina, bifurcación a subrutina, ramificación o *link*. Nos referiremos a la rutina que contiene la llamada a una subrutina como llamada a la subrutina. La instrucción de llamada a la subrutina tiene un solo campo de direcciones y realiza dos operaciones. Primero, almacena el valor del *PC*, que es la dirección siguiente a la instrucción de llamada a la subrutina, en una posición temporal. Esta dirección se llama *dirección de retorno*, y la instrucción correspondiente es el punto a continuación de la llamada a la subrutina. Segundo, la dirección en la instrucción de llamada a la subrutina, dirección de la primera instrucción de la subrutina, se carga en el *PC*. Cuando se accede a la siguiente instrucción, ésta procede de la llamada a la subrutina.

La última instrucción de cada subrutina debe ser una instrucción de retorno. La instrucción de retorno toma la dirección que fue almacenada por la instrucción de llamada a la subrutina y la coloca en el *PC*. Esto da lugar a un regreso al programa principal al siguiente punto donde se hizo la llamada a la subrutina.

Los procesadores utilizan diferentes posiciones temporales para almacenar la dirección de retorno. Algunos la almacenan en una posición fija de memoria, otros la almacenan en un registro del procesador, y otros en una pila. La ventaja de utilizar una pila para la dirección de retorno es que, cuando se llaman a una sucesión de subrutinas, la dirección correspondiente de retorno se coloca en la pila. Las instrucciones de retorno sacan la dirección de la posición más alta de la pila y su contenido lo transfieren al *PC*. De esta forma, un retorno se hace siempre al programa que llamó por última vez a la subrutina. Una instrucción de llamada a subrutina que utiliza una pila se realiza con las siguientes microoperaciones:

$SP \leftarrow SP - 1$	Decrementa el puntero de pila
$M[SP] \leftarrow PC$	Almacena la dirección de retorno en la pila
$PC \leftarrow$ Dirección efectiva	Transferencia del control a la subrutina

La instrucción de retorno se lleva a cabo sacando la dirección de retorno de la pila y la coloca en el *PC*:

$PC \leftarrow M[SP]$	Transfiere la dirección de retorno al *PC*
$SP \leftarrow SP + 1$	Incrementa el puntero de pila

Usando una pila para gestionar las subrutinas, todas las direcciones de retorno se almacenan automáticamente, mediante hardware, en la memoria de pila. De esta forma, el programador no tiene que tener en consideración la gestión de las direcciones de retorno de las subrutinas llamadas.

11-9 INTERRUPCIONES

Una interrupción de programa se usa para manejar diversas situaciones que necesitan una salida de la secuencia normal del programa. Una interrupción transfiere el control del programa que se está ejecutando a otro programa auxiliar, rutina de atención a la interrupción, como consecuencia de una petición generada externa o internamente. El control vuelve al programa original después de que la rutina de atención a la interrupción se ha ejecutado. Básicamente, las rutinas de atención a las interrupciones son similares a las subrutinas vistas anteriormente, excepto en tres aspectos:

1. La interrupción se inicia en un punto impredecible del programa por una señal externa o interna, en lugar de la ejecutarse por una instrucción.
2. La dirección de la rutina de atención que procesa la petición de interrupción se determina por el hardware, en lugar de un campo de dirección de una instrucción.
3. En respuesta a una interrupción, es necesario almacenar la información que contienen todos o algunos de los registros del procesador, en lugar de almacenar solamente el contador del programa.

Después de que el procesador ha sido interrumpido y la rutina de atención a la interrupción se ha ejecutado, el procesador debe volver exactamente al mismo estado que tenía antes de ser interrumpido. Sólo si ocurre esto, el programa podrá seguir ejecutándose como si nada hubiese sucedido. El estado del procesador al final de la ejecución de una instrucción se determina a partir del contenido del conjunto de registros. El *PSR*, además de contener información de los bits de status, puede especificar qué interrupciones están permitidas y si el sistema está en modo usuario o en modo supervisor. La mayoría de los procesadores tienen residente un sistema operativo que controla y supervisa al resto de los programas. Si el procesador está ejecutando un programa que es parte del sistema operativo, el procesador se coloca en modo supervisor, en el que ciertas instrucciones tienen privilegios y sólo se pueden ejecutar en modo supervisor. El procesador está en modo usuario cuando se ejecutan programas de usuario, en cuyo caso no se pueden ejecutar las instrucciones con privilegios. El modo del procesador en un instante dado se determina a partir de un bit o bits especiales del *PSR*.

Algunos procesadores sólo almacenan el contador de programa cuando responden a una interrupción. En dichos procesadores, el programa que realiza el procesado de los datos para dar servicio a la interrupción debe incluir instrucciones para almacenar los contenidos importantes del conjunto de registros. Otros procesadores almacenan el contenido de todos sus registros automáticamente en respuesta a una interrupción. Algunos procesadores tienen dos conjuntos de registros, de forma que cuando el programa conmuta del modo usuario al modo supervisor, como respuesta a una interrupción, no es necesario almacenar el contenido de los registros del procesador puesto que cada modo utiliza su propio conjunto de registros.

El procedimiento hardware para procesar interrupciones es muy similar a la ejecución de una instrucción de llamada a subrutina. El contenido del conjunto de los registros del procesador se almacenan temporalmente en memoria, típicamente en la memoria de pila, y la dirección de la primera instrucción de la rutina de atención a la interrupción se almacena en el PC. La dirección de la rutina de atención a la interrupción la escoge el hardware. Algunos procesadores

asignan una posición de memoria para guardar la dirección de la rutina de atención a la interrupción: la rutina de atención debe determinar la fuente de interrupción y proceder a atenderla. Otros procesadores asignan una posición de memoria aparte para cada una de las posibles fuentes de interrupción. A veces, la propia fuente de interrupción hardware proporciona la dirección de comienzo de la rutina de atención. En cualquier caso, los procesadores deber tener algún tipo de procedimiento hardware para seleccionar la dirección de la rutina de atención.

La mayoría de los procesadores no responderán a una interrupción hasta que la instrucción que se está ejecutando no se termine de ejecutar. Luego, justo antes de acceder a la siguiente instrucción, el control comprueba si ha habido alguna señal de interrupción. Si ha ocurrido, el control pasa a un ciclo de interrupción hardware. Durante este ciclo, el contenido de algunos registros o de todos se colocan en la pila. La dirección de comienzo de una interrupción concreta se transfiere al *PC*, y el control accede a la siguiente instrucción, con la que se inicia la rutina de atención a esa interrupción. La última instrucción de la rutina es una instrucción de retorno. Cuando este retorno se ejecuta, se extrae de la pila la dirección de retorno, que se transfiere al *PC*, así como el resto de información del contenido del conjunto de registro que se había almacenado, que se repone en los registros correspondientes.

Tipos de interrupciones

Los principales tipos de interrupciones que provocan una ruptura en la ejecución normal de un programa son las siguientes:

1. Interrupciones externas
2. Interrupciones internas
3. Interrupciones software

Las interrupciones externas proceden de dispositivos E/S, de dispositivos de temporización, de circuito que monitorizan la fuente de alimentación o de cualquier fuente externa. Las situaciones que provocan interrupciones son peticiones de dispositivos E/S que solicitan una transferencia de datos, dispositivos externos que acaban una transferencia de datos, la no finalización de un evento (en inglés *time- out*) o una amenaza de fallo de la alimentación. La interrupción de *time- out* puede ocurrir cuando hay un bucle sin fin que excede cierto tiempo. Una interrupción por fallo de la alimentación puede tener en su rutina de atención algunas instrucciones, pocas, que transfiera la información de todos los registros en una memoria no volátil, como un disco magnético, en pocos milisegundos antes de que caiga la alimentación.

Las interrupciones internas surgen del uso erróneo de una instrucción o dato. A las instrucciones internas también se les llama *traps*. Ejemplos de estas interrupciones causadas por circunstancias internas son: *overflow* aritmético, un intento de división por cero, un código de operación no válido, un *overflow* de la memoria de pila o una violación de protección. Esta última se produce cuando hay un intento de acceso a un área de memoria a la que el programa en curso no puede acceder. Las rutinas de atención que procesan las interrupciones internas determinan las medidas correctivas que se deben tomar en cada caso.

Las interrupciones externas e internas son iniciadas por el hardware del procesador. Por el contrario, una interrupción software se inicia mediante la ejecución de una instrucción. La interrupción software es una instrucción especial de llamada que se comporta como una interrupción en lugar de una llamada a subrutina. Se puede utilizar por el programador para iniciar una interrupción en un punto concreto del programa. El uso típico de la interrupción software se asocia con una instrucción de llamada del sistema. Esta instrucción proporciona un medio para cambiar de modo usuario a modo supervisor. Ciertas operaciones dentro del procesador se

pueden efectuar mediante el sistema operativo sólo en modo supervisor. Por ejemplo, una subrutina compleja de entrada o salida se hace en modo supervisor. Por el contrario, un programa escrito por un usuario debe correr en modo usuario. Cuando hay una petición de transferencia de E/S, el programa de usuario provoca una interrupción software, la cual almacena el contenido del *PSR* (con el bit de modo puesto en «usuario»), carga el nuevo contenido del *PSR* (con el bit de modo en «supervisor»), e inicia la ejecución de un programa del supervisor. La llamada del programa debe pasar información al sistema operativo para especificar la tarea concreta que se está solicitando.

Otro término alternativo para interrupción es *excepción*, que se puede aplicar sólo a las interrupciones internas o a todas las interrupciones, dependiendo del fabricante del procesador. Como ilustración del uso de los dos términos, a lo que un programador llama rutina de manejo de interrupciones puede llamarse como rutina de manejo de excepciones por otro programador.

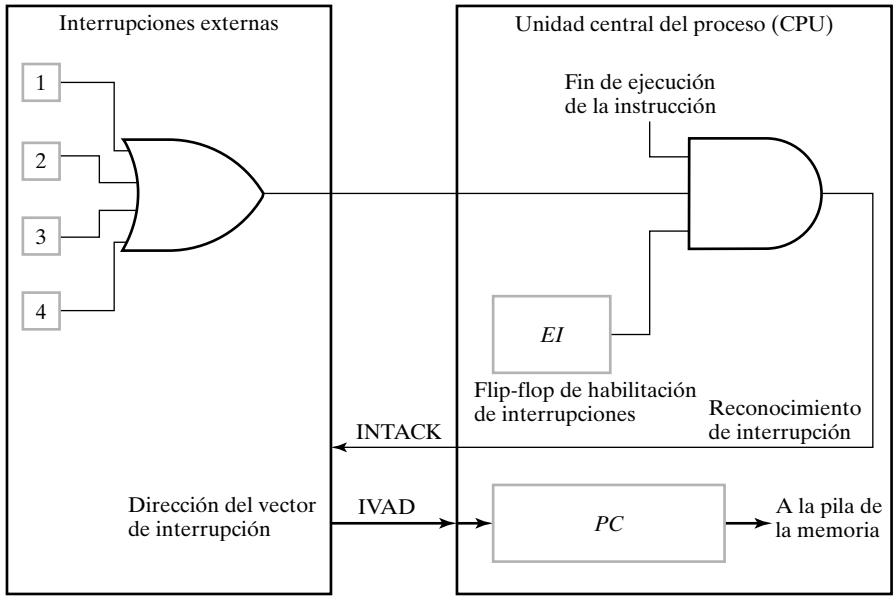
Procesamiento de interrupciones externas

Las interrupciones externas pueden tener una o más líneas de entrada. Si hay en el procesador más fuentes de interrupción que entradas de interrupciones, dos o más de esas fuentes están conectadas a una puerta OR para formar una línea común. Una señal de interrupción puede originarse en cualquier momento durante la ejecución del programa. Para asegurar que no se pierde información, el procesador suele reconocer la interrupción sólo después de que la instrucción en curso se ha terminado y sólo si el estado del procesador lo autoriza.

En la Figura 11-9 se muestra una configuración simplificada de una interrupción externa. Las cuatro fuentes de interrupciones externas se conectan a una puerta OR para formar una única señal de interrupción. Dentro de la CPU hay un flip-flop de habilitación de interrupciones (*EI*) que se puede poner a 0 o a 1 con dos instrucciones del programa: habilita interrupción (ENI, del inglés *enable interrupt*) y deshabilita interrupción (DSI, del inglés *disable interrupt*). Cuando *EI* es 0, se ignora la señal de interrupción. Si *EI* es 1 y la CPU está al final de la ejecución de una instrucción, el procesador reconoce la interrupción habilitando la salida de reconocimiento de interrupción *INTACK*. La fuente de interrupción responde a *INTACK* proporcionando la dirección del vector de interrupción *IVAD* a la CPU. El flip-flop controlado por programa, *EI*, permite al programador decidir si permite el uso de interrupciones. Si se inserta en el programa la instrucción *DSI* para poner a cero a *EI*, quiere decir que el programador no desea que el programa sea interrumpido. La ejecución de una instrucción ENI para poner a 1 a *EI*, índica que se permiten las interrupciones mientras el programa está corriendo.

El procesador responde a una señal de petición de interrupción si $EI = 1$ y la ejecución de la presente instrucción se ha completado. Las microinstrucciones típicas que realizan la interrupción son las siguientes:

$SP \leftarrow SP - 1$	Decrementa el puntero de pila
$M[SP] \leftarrow PC$	Almacena la dirección de retorno en la pila
$SP \leftarrow SP - 1$	Decrementa el puntero de pila
$M[SP] \leftarrow PSR$	Almacena el status de procesador en la pila
$EI \leftarrow 0$	Pone a 0 el flip-flop
$INTACK \leftarrow 1$	Habilita el reconocimiento de la interrupción
$PC \leftarrow IVAD$	Transfiere la dirección del vector de interrupción al PC Pasa a la fase de acceso



□ **FIGURA 11-9**
Ejemplo de configuración de interrupciones externas

La dirección de retorno disponible en el *PC* se introduce en la pila y el contenido del *PSR* también. *EI* se pone a 0 para deshabilitar posteriores interrupciones. El programa que atiende a la interrupción puede poner a 1 a *EI* con una instrucción siempre que se desee habilitar otras interrupciones. La CPU supone que la fuente externa proporcionará un *IVAD* como respuesta a un *INTACK*. El *IVAD* se toma como dirección de la primera instrucción de la rutina que atiende a la interrupción. Obviamente, se debe escribir un programa para este propósito y almacenarlo en la memoria.

El retorno de una interrupción se realiza con una instrucción al final de la rutina de atención, que es similar al retorno de una subrutina. La pila se vacía, y la dirección de retorno se transfiere al *PC*. Como el flip-flop *EI* se suele incluir en el *PSR*, el valor original de *EI* se repone cuando se carga el antiguo valor de *PSR*. Así, el sistema de interrupciones se habilita o deshabilita en el programa original, según estuviese antes de ocurrir la interrupción.

11-10 RESUMEN

En este capítulo hemos definido los conceptos de arquitectura de conjunto de instrucciones y los componentes de una instrucción y hemos explorado los efectos en los programas con el mayor número de direcciones por instrucción, utilizando tanto direcciones de memoria como direcciones de registro. Esto conduce a la definición de cuatro tipos de arquitecturas de direccionamiento: memoria a memoria, registro a registro, acumulador único y pila. Los modos de direccionamiento especifican cómo se interpreta la información de unas instrucciones para determinar la dirección efectiva de un operando.

Los procesadores de reducido conjunto de instrucciones (RISC) y de conjunto de instrucciones complejas (CISC) son dos categorías habituales de arquitecturas de conjunto de instrucciones. Un RISC tiene como objetivo alto rendimiento y alta velocidad de ejecución. Por el contra-

rio, un CISC intenta acercarse a las operaciones utilizadas en los lenguajes de programación y facilitar programas compactos.

Las tres categorías de instrucciones básicas son transferencia de datos, manipulación de datos y control de programa. Para elaborar las instrucciones de transferencia de datos aparece el concepto de memoria de pila. Las transferencias entre la CPU y E/S se direccionan mediante dos métodos diferentes: E/S independiente, con espacio de direcciones separado, y E/S ubicada en memoria, que utiliza parte del espacio de la memoria. Las instrucciones de manipulaciones de datos se clasifican en tres clases: aritméticas, lógicas y de desplazamiento. El formato en punto flotante y sus operaciones manejan rangos de valores más amplios para los operandos de las operaciones aritméticas.

Las instrucciones de control de programa incluyen la transferencia básica del control incondicional y condicional, esta última puede o no usar códigos de condición. Las llamadas a subrutinas y los retornos permiten que se pueda romper la secuencia de los programas para realizar tareas útiles. La interrupción de la secuencia normal de ejecución de un programa se basa en tres tipos de interrupciones: externas, internas y software. También se les llama excepciones a las interrupciones que necesitan acciones de procesamiento especiales en la inicialización de rutinas de atención para atenderlas y en el retorno de la excepción de los programas interrumpidos.

REFERENCIAS

1. MANO, M. M.: *Computer Engineering: Hardware Design.* Englewood Cliffs, NJ: Prentice Hall, 1988.

2. GOODMAN, J., and K. MILLER: *A Programmer's View of Computer Architecture*. Fort Worth, TX: Saunders College Publishing, 1993.

3. HENNESSY, J. L., and D. A. PATTERSON: *Computer Architecture: A Quantitative Approach*, 2nd Ed. San Francisco, CA: Morgan Kaufmann, 1996.

4. MANO, M. M.: *Computer System Architecture*, 3rd Ed. Englewood Cliffs, NJ: Prentice Hall, 1993.

5. PATTERSON, D. A., and J. L. HENNESSY: *Computer Organization and Design: The Hardware/Software Interface*, 2nd Ed. San Mateo, CA: Morgan Kaufmann, 1998.

6. *IEEE Standard for Microprocessor Assembly Language*. (IEEE Std 694- 1985.) New York, NY: The Institute of Electrical and Electronics Engineers.

7. *IEEE Standard for Binary Floating-Point Arithmetic*. (ANSI/IEEE Std 754-1985.) New York, NY: The Institute of Electrical and Electronics Engineers.

PROBLEMAS

El signo (+) indica problemas más avanzados y el asterisco (*) indica que hay una solución disponible en la dirección de Internet: http://www.librosite.net/Mano.

11-1. Basándose en las operaciones ilustradas en la Sección 11-1, escriba un programa que evalúe la expresión aritmética:

$$X = (A - B) \times (A + C) \times (B - D)$$

Haga efectivo el uso de los registros para minimizar el número de instrucciones MOV y LD donde sea posible.

(a) Suponga una arquitectura registro a registro con instrucciones con tres direcciones.
(b) Suponga una arquitectura memoria a memoria con instrucciones con dos direcciones.
(c) Suponga un procesador con un solo acumulador con instrucciones con una dirección.

11-2. *Repita el Problema 11-1 para

$$Y = (A + B) \times C \div (D - (E \times F))$$

Todos los operando están inicialmente en la memoria y DIV representa dividir.

11-3. *Se ha escrito un programa para una arquitectura con pila para evaluar la expresión aritmética

$$X = (A - B) \times (A + C) \times (B - D)$$

(a) Encuentre la correspondiente expresión en NPI.
(b) Escriba el programa utilizando las instrucciones PUSH, POP, ADD, MUL, SUB y DIV.
(c) Muestre el contenido de la pila después de la ejecución de cada instrucción.

11-4. Repita el Problema 11-3 para la expresión aritmética

$$(A + B) \times C \div (D - (E \times F))$$

11-5. Una instrucción de dos palabras está almacenada en la memoria en la dirección designada por el símbolo *W*. El campo de direcciones de la instrucción (almacenado en $W + 1$) se designa con el símbolo *Y*. El operando utilizado durante la ejecución de la instrucción se almacena en la dirección simbolizada mediante *Z*. Un registro índice contiene el valor *X*. Exponga cómo se calcula *Z* a partir de otras direcciones si el modo de direccionamiento de la instrucción es:

(a) Directo. **(b)** Indirecto. **(c)** Relativo. **(d)** Indexado.

11-6. *Una instrucción de bifurcación de modo relativo de dos palabras se almacena en la posición 207 y 208 (decimal). La bifurcación se hace en una dirección equivalente a 195 (decimal). Designe el campo de dirección de la instrucción (almacenado en la dirección 208) como *X*.

(a) Determine el valor de *X* en decimal.
(b) Determine el valor de *X* en binario, usando 16 bits. (Véase que el número es negativo y debe estar en complemento a 2 ¿Por qué?)

11-7. Repita el Problema 11-6 para una instrucción de ramificación en la posición 143 y 144 y una dirección de bifurcación equivalente a 1000. Todos los valores están en decimal.

11-8. Cuántas veces hace referencia la unidad de control a la memoria cuando accede y ejecuta una instrucción de dos palabras con modo de direccionamiento indirecto si la instrucción es:

(a) Un cálculo que requiere un operando de una posición de memoria que devuelve el resultado a la misma posición.
(b) Una bifurcación

11-9. Una instrucción está almacenada en la posición 300 con su campo de direcciones en la posición 301. El campo de direcciones tiene el valor 211. Un registro del procesador, $R1$, contiene el número 189. Evalúe la dirección efectiva si el modo de direccionamiento de la instrucción es:

(a) Directo.
(b) Inmediato.
(c) Relativo.
(d) Registro indirecto.
(e) Indexado con $R1$ como registro índice.

11-10. *Un procesador tiene una palabra de 32 bits de longitud y todas sus instrucciones tienen una palabra de longitud. El banco de registros del procesador tiene 16 registros.

(a) Para un formato sin campo de modo y con tres direcciones de registro. ¿Cuál es el número máximo de códigos de operación posibles?
(b) Para un formato con dos campos de direcciones de registros, un campo de memoria y un máximo de 100 códigos de operación. ¿Cuál es el número máximo de bits de direcciones de memoria disponible?

11-11. Un procesador, con un banco de registros y sin instrucciones PUSH y POP, se utiliza para hacer una pila. El procesador tiene los siguientes modos indirectos con registros:

Registro indirecto + incremento:

LD R2 R1	$R2 \leftarrow M[R1]$
	$R1 \leftarrow R1 + 1$
ST R2 R1	$M[R1] \leftarrow R2$
	$R1 \leftarrow R1 + 1$

Decremento + registro indirecto:

LD R2 R1	$R1 \leftarrow R1 - 1$
	$R2 \leftarrow M[R1]$
ST R2 R1	$R1 \leftarrow R1 - 1$
	$M[R1] \leftarrow R2$

Muestre cómo estas instrucciones se pueden utilizar para efectuar las instrucciones PUSH y POP y usando el registro $R6$ como puntero de pila.

11-12. Una instrucción compleja, *push registers* (PSHR), pone el contenido de todos los registros en una pila. Hay ocho registros, $R0$ a $R7$, en la CPU. La instrucción POPR, saca el contenido de los registros de la pila colocándolos de nuevo en sus registros correspondientes.

(a) Describa una posible transferencia de registros en la ejecución de PSHR.
(b) Describa una posible transferencia de registros para la ejecución de POPR.

11-13. Un procesador con sistema independiente de E/S tiene la instrucción de entrada y salida

IN R[DR] ADRS

OUT ADRS R[SB]

donde ADRS es la dirección de un puerto de E/S con registro. Señale las instrucciones equivalentes para un procesador con la E/S ubicada en memoria.

11-14. *Suponga un procesador con palabras de 8 bits con suma de precisión múltiple de dos números de 32 bits sin signo,

1F C6 24 7B + 00 57 ED 4B

(a) Escriba un programa que ejecute la suma utilizando instrucciones de suma y suma con acarreo.
(b) Ejecute el programa para los operandos dados. Cada byte se expresa con números de 2 dígitos hexadecimales.

11-15. Realice las operaciones lógicas AND, OR y XOR de los dos bytes 00110101 y 10111001.

11-16. Dado el valor de 16 bits 1010 1001 0111 1100 ¿Qué operación se debe efectuar y qué operandos se necesitan para

(a) poner a uno los 8 bits menos significativos?
(b) complementar los bits de las posiciones impares (la más a la izquierda es la 15 y la más a la derecha es la 0)?
(c) poner a cero los bits de las posiciones impares?

11-17. *Un registro de 8 bits contiene el valor 01101001 y el bit de acarreo igual a 1. Realice las operaciones de desplazamiento dadas mediante las instrucciones enumeradas en la Tabla 11-5 como una secuencia de operaciones sobre estos registros.

11-18. Muestre cómo los siguientes números en punto flotante se han de sumar para obtener un resultado normalizado:

$$(-.12345 \times 10^{+3}) + (+.71234 \times 10^{-1})$$

11-19. *Un número de 36 bits en punto flotante tiene 26 bits más signo para la mantisa y 8 bits más signo para el exponente. ¿Cuáles son las cantidades positivas distintas de cero más grandes y más pequeñas para números normalizados?

11-20. *Un exponente de 4 bits utiliza un número en exceso a 7 para el sesgo. Enumere los exponentes sesgados en binario desde 18 hasta 27.

11-21. El estándar IEEE para doble precisión para operandos en punto flotante es de 64 bits. El signo ocupa un bit, el exponente tiene 11 bits y la mantisa 52 bits. El exponente sesgado es 1023 y la base es 2. Hay un bit implícito a la derecha del punto binario de la mantisa. Infinito se representa con el exponente sesgado igual a 2047 y la mantisa igual a 0.

(a) Indique la fórmula para encontrar el valor decimal de un número normalizado.
(b) Enumere algunos exponentes con sesgo en binario, como se hizo en la Tabla 11-6.
(c) Calcule los números positivos más grande y más pequeño que se pueden representar.

11-22. Demuestre que si la igualdad $2^x = 10^y$ se cumple, entonces $y = 0.3x$. Utilizando esta relación, calcule los números normalizados en punto flotante más grande y más pequeño en decimal que se pueden representar con el formato IEEE de precisión simple.

11-23. *Se necesita bifurcar a ADRS si el bit menos significativo de un operando en un registro de 16 bits es 1. Muestre cómo se puede hacer con las instrucciones TEST (Tabla 11-7) y BNZ (Tabla 11-8).

11-24. Considere los dos números de 8 bits $A = 00101101$ y $B = 01101001$.

(a) Dé el valor decimal equivalente para cada número suponiendo que (1) son sin signo y (2) son con signo en complemento a 2.
(b) Sume los dos números binarios e interprete la suma suponiendo que los números son (1) sin signo y (2) con signo en complemento a 2.
(c) Determine los valores de *C* (acarreo), *Z* (cero), *N* (signo) y *V* (*overflow*) de los bits de status después de la suma.
(d) Enumere las instrucciones de bifurcación condicional de la Tabla 11-8 que tendrá una condición de verdad (*true*).

11-25. *Un programa en un procesador compara dos números sin signo *A* y *B* realizando la resta $A - B$ y actualizando los bits de status.
Sea $A = 01011101$ y $B = 01011100$.

(a) Evalúe la diferencia e interprete el resultado binario.
(b) Determine los valores de los bits de status *C* (acarreo) y *Z* (cero).
(c) Enumere las instrucciones de ramificación de la Tabla 11-9 que tendrán una condición *true*.

11-26. Un programa en un procesador compara dos números en complemento a 2, *A* y *B*, realizando la substracción $A - B$ y actualizando los bits de status.
Sea $A = 11011110$ y $B = 11010110$.

(a) Evalúe la diferencia e interprete el resultado binario.
(b) Determine los valores de los bits de status *N* (signo), *Z* (cero) y *V* (*overflow*).
(c) Enumere las instrucciones de bifurcación condicional de la Tabla 11-10 que tendrán la condición *true*.

11-27. *La posición más alta de la memoria de pila contiene el valor 30000. El puntero de pila *SP* contiene 2000. Una instrucción de llamada a subrutina de dos palabras está ubicada en la dirección 2000, seguida de un campo de direcciones con el valor 0301 en la posición 2001. ¿Cuál es el valor del *PC*, *SP* y la posición más alta de la pila

(a) antes acceder a la instrucción de llamada a subrutina de la memoria?
(b) después de ejecutar la instrucción de llamada a subrutina?
(c) después de volver de la subrutina?

11-28. Un procesador no tiene pila pero en su lugar utiliza el registro *R*7 como un registro de enlace (*link*), es decir, almacena la dirección de retorno en *R*7.

(a) Muestre la transferencia de registros para una instrucción de bifurcación y de *link*.
(b) Suponiendo que existen otra bifurcación y otro *link* en la subrutina invocada ¿Qué acción debe realizar el software antes de que suceda la bifurcación y el *link*?

11-29. ¿Cuáles son las diferencias básicas entre una bifurcación, una llamada a subrutina y una interrupción de programa?

11-30. *Indique cinco ejemplos de interrupciones externas y cinco ejemplos de interrupciones internas. ¿Cuál es la diferencia entre una interrupción hardware y una llamada a subrutina?

11-31. Un procesador responde a una señal de petición de interrupción introduciendo en la pila el contenido del *PC* y de *PSR*. El procesador lee después el nuevo contenido de *PSR* de una posición de memoria dada mediante el vector de dirección (*IVAD*). La primera dirección de la rutina de atención a la interrupción se toma de la posición *IVAD* + 1.

(a) Enumere la secuencia de microoperaciones que realiza la interrupción.

(b) Enumere la lista de secuencia de microoperaciones que realiza el retorno de la interrupción.

CAPÍTULO

12

Unidades centrales de procesamiento RISC y CISC

La Unidad Central de Procesamiento (*Central Processing Unit*, CPU) es el componente clave de una computadora digital. Su propósito es decodificar las instrucciones recibidas de la memoria y realizar operaciones de transferencia, aritméticas, lógicas y de control con los datos almacenados en los registros internos, memoria o unidades de interfaz de E/S. Externamente, la CPU tiene uno o más buses para la transferencia de instrucciones, datos e información de control con los componentes a los que está conectado.

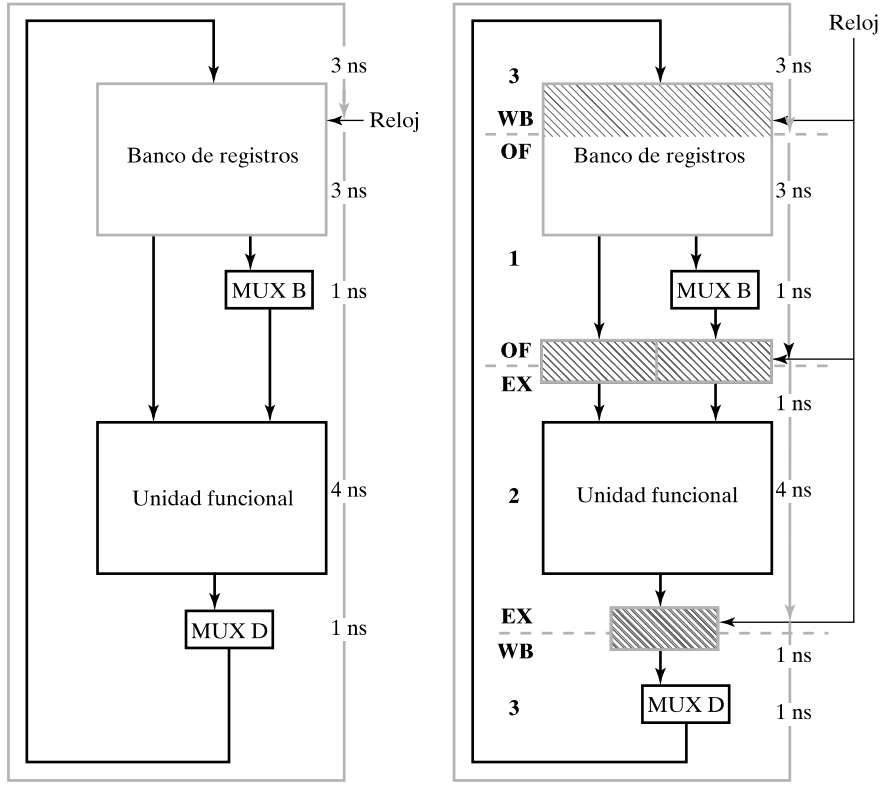
En la computadora genérica al comienzo del Capítulo 1, la CPU es una parte del procesador. Sin embargo, las CPUs pueden aparecen en otros sitios, aparte de los computadoras. Procesadores pequeños y relativamente sencillos, llamados microcontroladores, se usan en computadoras y en otros sistemas digitales para realizar tareas concretas o especializadas. Por ejemplo, hay un microcontrolador en el teclado y en el monitor de una computadora genérica. En dichos microcontroladores, las CPUs pueden ser bastante diferentes de las presentadas en este capítulo. Las longitudes de las palabras pueden ser cortas (por ejemplo, de ocho bits), el número de registros pequeño y el conjunto de instrucciones limitado. El rendimiento, relativamente hablando, es bajo pero adecuado. Lo más importante es que el coste de estos microcontroladores es muy bajo, haciendo su uso muy atractivo.

El estudio de este capitulo se hace sobre y paralelo al del Capítulo 10. Comienza convirtiendo la ruta de datos del Capítulo 10 en una ruta de datos segmentada (en *pipeline*). Se añade una unidad de control segmentada para crear un procesador de conjunto de instrucciones reducido (*reduced instruction set computer*, RISC) que es similar a la de un procesador de un solo ciclo. Se presentan los problemas que surgen debido al uso de la segmentación y su solución en el contexto del diseño de RISC. Después, se aumenta la unidad de control para tener un procesador de conjunto de instrucciones complejo (*complex instruction set computer*, CISC) que es similar al procesador multiciclo. Se presenta una visión global de algunos métodos utilizados para mejorar el rendimiento de un procesador segmentado o en *pipeline*. Para terminar, relacionaremos las ideas discutidas para el diseño de sistemas digitales en general.

12-1 RUTA DE DATOS SEGMENTADA

La Figura 10-17 se usó para ilustrar el camino de mayor retardo presente en un procesador de un solo ciclo y el límite de la frecuencia de reloj resultante. Con un enfoque más cercano, la Figura 12-1(a) ilustra los valores máximos de los retardos para cada uno de los componentes de una ruta de datos típica. Se necesita un máximo de 4 ns (3 ns + 1 ns) para leer dos operandos del banco de registros o para leer un operando del banco de registros y obtener una constante de MUX *B*. También se necesita un máximo de 4 ns (1 ns + 3 ns) para escribir el resultado de nuevo en el banco de registros, incluyendo el retardo de MUX *D*. Sumando estos retardos, encontramos que se necesitan 12 ns para realizar una sola microoperación. La tasa máxima a la que las microoperaciones se pueden realizar es la inversa de 12 ns (es decir, 83.3 MHz). Esta es la frecuencia máxima a la que el reloj puede funcionar ya que 12 ns es el periodo de reloj más pequeño que permitirá que cada microoperación se complete con certeza. Como se ilustra en la Figura 10-17, el retardo de los caminos que pasan a través de la ruta de datos y de la unidad de control limita la frecuencia de reloj incluso a un valor más pequeño. Sólo para la ruta de datos y para la combinación de la ruta de datos y de la unidad de control de un procesador de un solo ciclo, la ejecución de una microoperación constituye la ejecución de una instrucción. Así, la tasa de ejecución de instrucciones es igual a la frecuencia de reloj.

Suponga ahora que la tasa de ejecución de la ruta de datos no es la adecuada para una cierta aplicación y que no tenemos componentes más rápidos con los que reducir los 12 ns necesarios



□ **FIGURA 12-1**
Temporización de la ruta de datos

para completar una microoperación. Aún así, puede ser posible reducir el periodo del reloj e incrementar su frecuencia. Esto se puede hacer dividiendo el camino con retardo de 12 ns con registros. A la ruta de datos resultante, mostrada en la Figura 12-1(b) se le llama ruta de datos segmentada o ruta de datos en *pipeline*, o simplemente *pipeline*.

Los tres conjuntos de registros dividen el retardo de la ruta de datos original en tres partes. Estos registros se muestran sombreados en azul. El banco de registros contiene el primer conjunto de registros. El sombreado sólo cubre la mitad superior del banco de registros ya que la mitad inferior se considera como la lógica combinacional que selecciona los dos registros que pueden ser leídos. Los dos registros que almacenan el dato *A* del banco de registros y la salida de MUX *B* forman el segundo conjunto de registros. El tercer conjunto de registros almacena las entradas de MUX *D*.

El termino «*pipeline*» (en español tubería), desafortunadamente, no proporciona la mejor analogía para la estructura correspondiente de una ruta de datos. Una mejor analogía para la ruta de datos en *pipeline* es una línea de producción. Un ejemplo habitual de una línea de producción es una estación de lavado automático de coches en el que los coches pasan a través de una serie de puestos en los que se realiza una función en particular de lavado:

1. Lavado - Lavado con agua caliente y jabón,
2. Enjuague - aclarado con agua tibia y,
3. Secado - soplado de aire sobre la superficie

En este ejemplo, el procesado de un vehículo a través del lavado está compuesto por tres pasos y requiere de una cierta cantidad de tiempo para llevarlos a cabo. Usando esta analogía, el procesado de una instrucción mediante un *pipeline* está formado por $n > 2$ pasos y cada paso necesita una cierta cantidad de tiempo para llevarse a cabo. La cantidad de tiempo necesario para procesar una instrucción se llama *tiempo de latencia*. Usando la analogía con la estación de lavado de coches, el tiempo de latencia es la cantidad de tiempo necesario para que el coche pase por los tres puestos que realizan los tres pasos del proceso. Este tiempo permanece igual independientemente de si hay un coche o si hay tres coches en la estación de lavado al mismo tiempo.

Continuando esta analogía de la ruta de datos en *pipeline* con la estación de lavado ¿Como sería una ruta de datos sin *pipeline*? Sería un lavado del coche con los tres pasos disponibles en una sola estación, donde los pasos se realizan en serie. Ahora podemos comparar las analogías, comparando de este modo la ruta de datos en *pipeline* y sin *pipeline*. Para una estación múltiple de lavado de coches y para una estación sencilla, la latencia es aproximadamente la misma. Por tanto, ir a una estación múltiple no decrementa el tiempo necesario para lavar un coche. Sin embargo, suponga que tenemos en cuenta la frecuencia a la que los coches ya lavados salen de los dos tipos de estaciones de lavado. En la estación sencilla, esta frecuencia es la inversa del tiempo de latencia. Por el contrario, en la estación múltiple de lavado, un coche sale ya lavado con una frecuencia de tres veces la inversa del tiempo de latencia. De esta forma, hay un factor de mejora de tres en la frecuencia o tasa de salida de coches lavados. Basándose en la analogía para las rutas de datos en *pipeline* con *n* etapas y una ruta de datos sin *pipeline*, la primera tiene una tasa de procesamiento o *throughput* para las instrucciones que es *n* veces la de la última. La estructura deseada, ruta de datos convencional y sin *pipeline* descrita en el Capítulo 10, se ilustra en la Figura 12-1(b). El acceso del operando (OF) se hace en la etapa 1, la ejecución (EX) en la etapa 2 y la escritura (WB) en la etapa 3. Estas etapas se etiquetan al lado con sus abreviaturas correspondientes. En este punto, la analogía se rompe algo ya que el coche se mueve suavemente a través de la estación de lavado mientras que el dato en el *pipeline* se mueve en sincronía con un reloj que controla el movimiento de una etapa a otra. Esto tiene algunas impli-

caciones interesantes. Primero, el movimiento del dato a través del *pipeline* se hace en pasos discretos en lugar de hacerlo de forma continua. Segundo, la cantidad de tiempo en cada etapa debe ser la del periodo del reloj y la misma para todas las etapas. Para tener un mecanismo que separe las etapas del *pipeline*, se colocan unos registros entre las etapas del *pipeline*. Estos registros dan un almacenamiento temporal para los datos que pasan a través de pipeline, y se les llama *registros del pipeline*.

Volviendo a la ruta de datos en *pipeline* del ejemplo de la Figura 12-1(b), la etapa 1 del *pipeline* tiene el retardo necesario para la lectura del banco de registros seguida de la selección de MUX *B*. Este retardo es de 3 más 1 ns. La etapa 2 del *pipeline* tiene 1 ns de retardo del registro más 4 ns de la unidad funcional, dando lugar a 5 ns. La etapa 3 tiene 1 ns del registro, el retardo de la selección de MUX *D* y el retardo de escribir de nuevo en el banco de registros. Este retardo es 1 + 1 + 3, un total de 5 ns. Así, todos los retardos de flip-flop a flip-flop son, como máximo, de 5 ns, permitiendo tener un periodo de reloj mínimo de 5 ns (asumiendo que el tiempo de *setup* de los flip-flops es cero) y una frecuencia máxima de reloj de 200 MHz, en comparación con los 83.3 MHz de la ruta de datos sencilla. Esta frecuencia de reloj se corresponde con el máximo *throughput* del *pipeline*, que es de 200 millones de instrucciones por segundo, en torno a 2.4 veces la de la ruta de dato sin *pipeline*. Incluso cuando hay tres etapas, el factor de mejora no es tres. Esto es debido a dos factores: (1) el retardo distribuido con los registros del *pipeline* y (2) las diferencias entre el retardo de la lógica asignada a cada etapa. El periodo de reloj se selecciona según el retardo más largo, en lugar del retardo medio asignado a cualquier etapa.

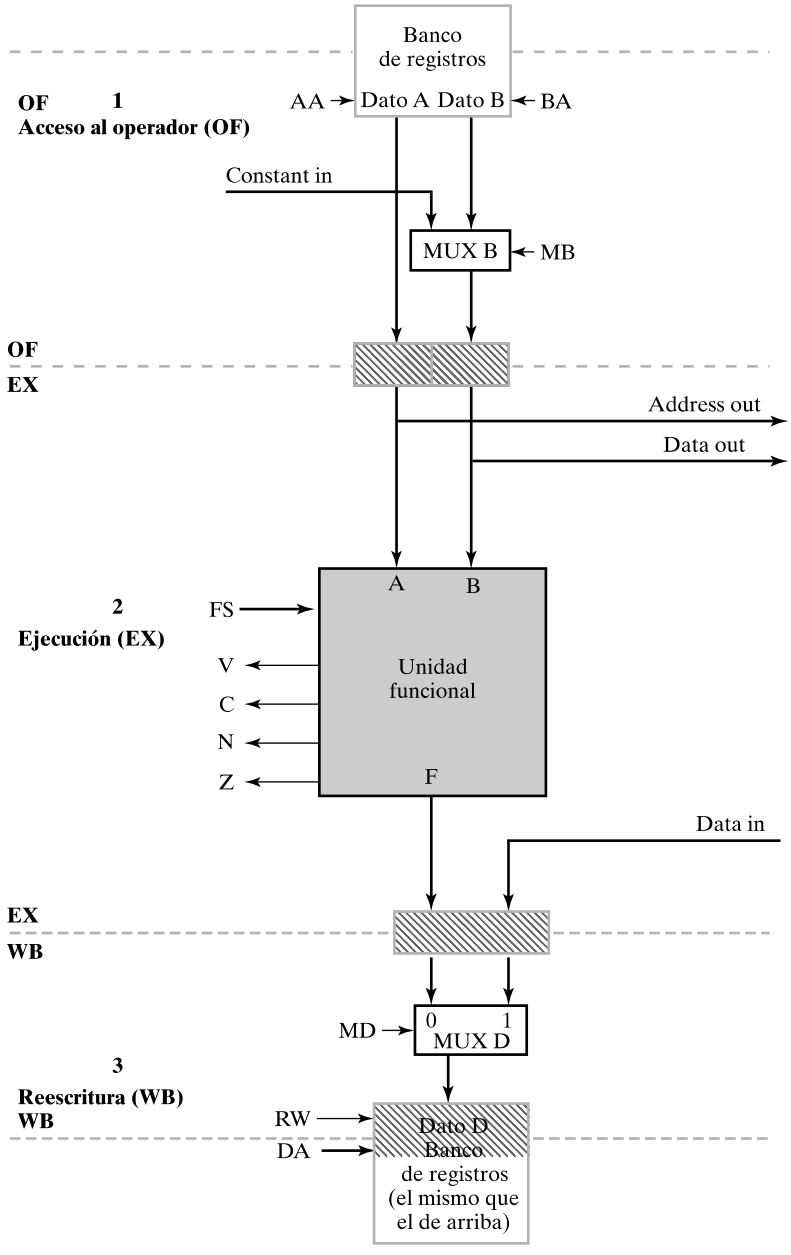
En la Figura 12-2 aparece un diagrama más detallado de la ruta de datos segmentada. En este diagrama, en lugar de mostrar la ruta desde la salida del MUX *D* a la entrada al banco de registros, el banco de registros se muestra dos veces, una en la etapa OF, donde se lee, y la otra en la etapa WB, donde se escribe.

La primera etapa, OF, es la etapa de acceso al operando. El acceso al operando consiste en la lectura de los valores del registro a utilizar del banco de registro y, para el Bus B, la selección entre un valor del registro o una constante utilizando el MUX *B*. A continuación de la etapa OF está el primer registro del *pipeline*. Los registros del *pipeline* almacenan el operando u operandos para utilizarlos en la siguiente etapa en el siguiente ciclo de reloj.

La segunda etapa del *pipeline* es la etapa de ejecución, llamada EX. En esta etapa se realiza una operación en la unidad funcional en la mayoría de las microoperaciones. El resultado producido en esta etapa se captura por el segundo registro del *pipeline*.

La tercera y última etapa del *pipeline* es la etapa de reescritura, llamada WB. En esta etapa, se selecciona el resultado almacenado en la etapa EX o el valor de *Data in* con el MUX *D* y se vuelve a rescribir en el banco de registro al final de la etapa. La etapa WB termina la ejecución de cada microoperación que necesita escribir en un registro.

Antes de abandonar la analogía con la estación de lavado, examinaremos el coste de una estación de lavado simple y la de tres etapas. Primero, aunque la estación de lavado lava los coches tres veces tan rápido como lo hace la estación sencilla, cuesta tres veces más en términos de espacio. Además, tiene un mecanismo que mueve el coche a lo largo de las etapas. De esta forma parece que no compensa mucho en coste comparado con tener tres estaciones sencillas ensambladas de tres etapas operando en paralelo. No obstante, desde un punto de vista comercial, se ha demostrado que compensa en términos de coste. En términos de coches lavados ¿Puede imaginar por qué? Por el contrario, para la ruta de datos en *pipeline*, los registros dividen una ruta de datos en tres partes. De esta forma, en una primera estimación de incremento de costes, éste se debe principalmente al uso de los registros del *pipeline*.



□ **FIGURA 12-2**
Diagrama de bloques de una ruta de datos segmentada

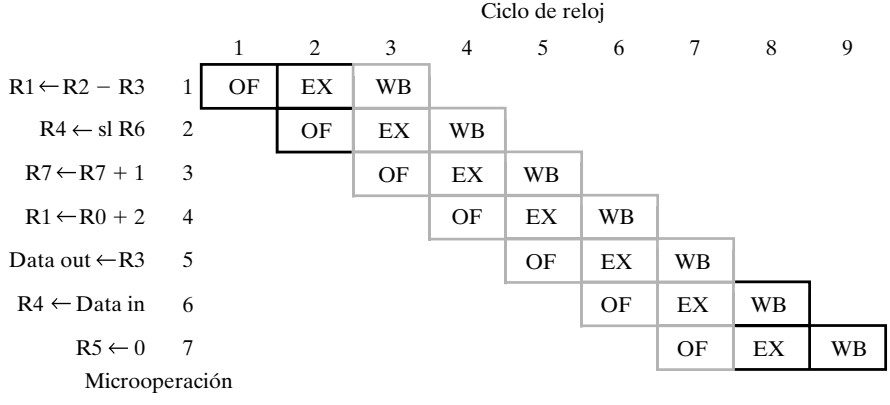
Ejecución de microoperaciones en *Pipeline*

Hay hasta tres operaciones en cualquier etapa que lava el coche en un momento dado. Por analogía, debería haber hasta tres microoperaciones en cualquier instante de tiempo en la ruta de datos segmentada.

Examinemos ahora la ejecución de esta secuencia de microoperaciones en relación a las etapas del *pipeline* de la Figura 12-2. En el ciclo de reloj 1, la microoperación 1 se realiza en la etapa OF. En el ciclo de reloj 2, la microoperación 1 está en la etapa EX y la microoperación 2 en la etapa OF. En el ciclo de reloj 3, la microoperación 1 está en la etapa WB, la microoperación 2 en la etapa EX y la microoperación 3 en la etapa OF. De esta forma, al final del tercer ciclo de reloj, la microoperación 1 se ha completado, la microoperación 2 se ha completado en dos terceras partes y de la microoperación 2 se ha realizado una tercera parte. Así que se han completado $1 + 2/3 + 1/3 = 2.0$ microoperaciones en tres ciclos de reloj, es decir, en 15 ns. En una ruta de datos convencional habríamos completado solamente la ejecución de la microoperación 1. De esto se concluye que el rendimiento de la ruta de datos en *pipeline* es superior en este ejemplo.

El procedimiento que hemos utilizado para analizar la secuencia de microoperaciones es, cuando menos, tedioso. Para terminar el análisis de la temporización de la secuencia utilizaremos un *diagrama de patrones de ejecución* del *pipeline*, como el que se muestra en la Figura 12-3. Cada posición vertical del diagrama representa la microoperación a realizar y cada posición horizontal representa un ciclo de reloj. Una posición del diagrama representa la etapa de procesamiento de la microoperación. Por ejemplo, la etapa de ejecución (EX) de la microoperación 4, que suma la constante 2 a *R*0 tiene lugar en el ciclo de reloj 5. Podemos ver del conjunto del diagrama que la secuencia de las siete microoperaciones necesita nueve ciclos de reloj para terminar su ejecución completamente. El tiempo necesario de ejecución de $9 \times 5 = 45$ ns comparado con los $7 \times 12 = 84$ ns de la ruta de datos convencional. De esta forma, la secuencia de microoperaciones se ejecuta en torno a 1.9 veces más rápidamente.

Vamos a examinar el patrón de ejecución cuidadosamente. En los dos primeros ciclos de reloj, no están activadas todas las etapas de *pipeline* puesto que éste se está llenando. En los siguientes cinco ciclos de reloj están activadas todas las etapa del *pipeline*, según se indica en trama, y el *pipeline* se está utilizando completamente. En los dos últimos ciclos de reloj no están activas todas las etapas del *pipeline* puesto que éste se está vaciando. Si queremos encontrar la mayor mejora de la ruta de datos segmentada sobre la convencional, compararemos las dos cuando el *pipeline* está lleno. En estos cinco ciclos de reloj, del 3 al 7, el *pipeline* ejecuta $(5 \times 3) \div 3 = 5$ microoperaciones en 25 ns. En el mismo tiempo, la ruta de datos convencional ejecuta $25 \div 12 = 2.083$ microoperaciones. De esta forma, la ruta de datos segmentada ejecuta, en el mejor de los casos $25 \div 2.083 = 2.4$ veces más operaciones, en un tiempo dado, que



□ **FIGURA 12-3**
Patrón de ejecucioín del *pipeline* para una secuencia de microoperaciones

la ruta de datos convencional. En esta situación ideal, decimos que el movimiento de datos (*throughput*) de la ruta de datos segmentada es 2.4 veces la de la convencional. Dese cuenta de que la velocidad de llenado y vaciado del *pipeline* está por debajo del valor máximo 2.4. Otros temas asociados con los *pipelines* —en particular, proporcionar una unidad de control a la ruta de datos segmentada y su comportamiento con los conflictos— se cubren en las dos secciones siguientes.

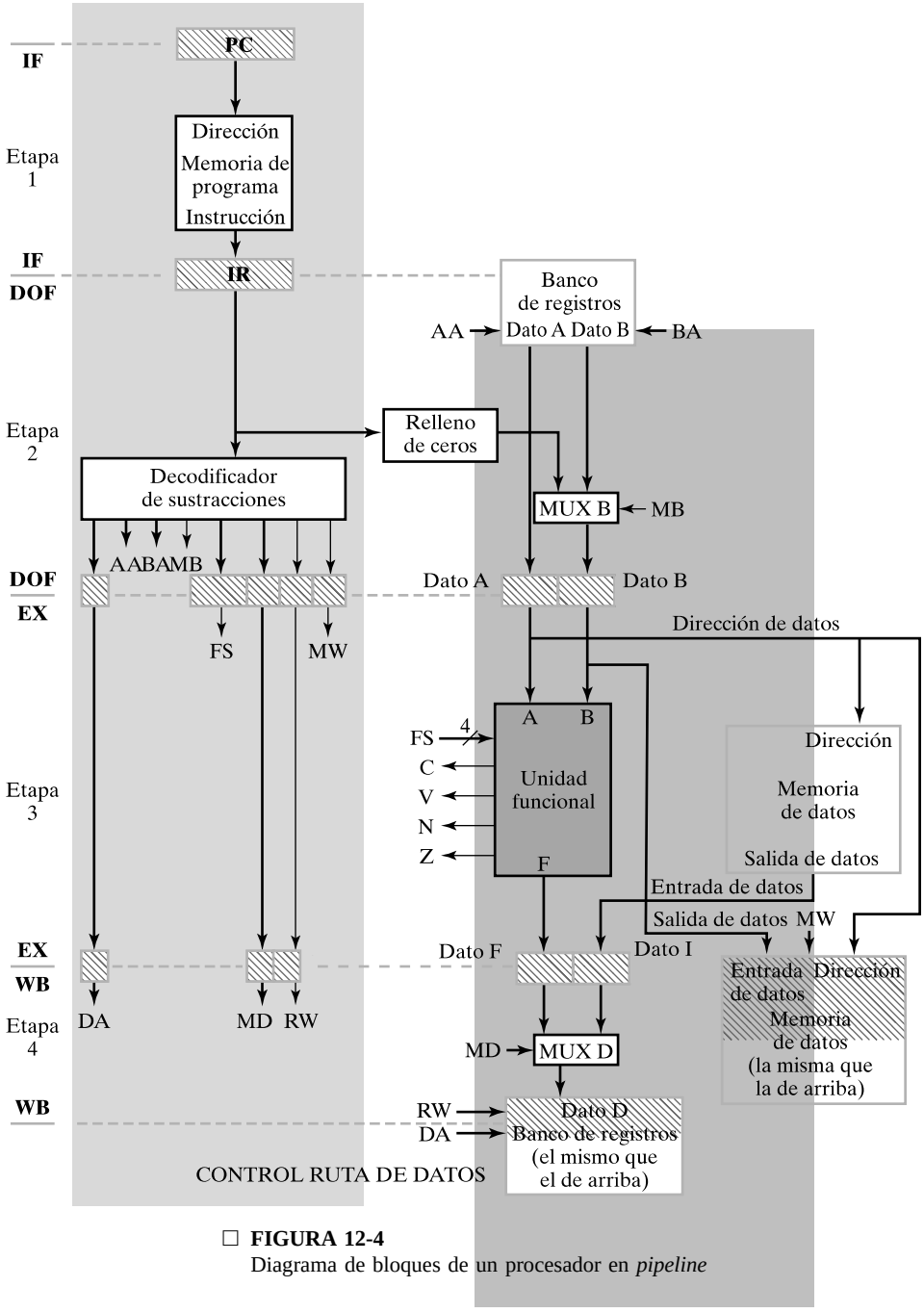
12-2 CONTROL DE LA RUTA DE DATOS SEGMENTADA

En esta sección, se especifica una unidad de control para realizar una CPU utilizando la ruta de datos de la sección anterior. Como las instrucciones se leen de la memoria según se van ejecutando, añadimos una etapa análogamente a la utilizada en la estación de lavado vista anteriormente. Análogamente al acceso de la instrucción de la memoria de instrucciones, las operaciones en la estación de lavado de coches se especifican mediante hojas de órdenes, producidas por un servidor, que permite variar las funciones realizadas en las etapas de la estación de lavado. La hoja de órdenes, que es análoga a una instrucción, acompaña al coche según avanza en la línea de lavado.

La Figura 12-4 muestra el diagrama de bloques de un procesador en *pipeline* basado en un procesador de un solo ciclo. La ruta de datos es la de la Figura 12-2. El control tiene una etapa adicional para el acceso a las instrucciones que incluye un *PC* y una memoria de instrucciones. Esto sería la etapa 1 del *pipeline* combinado. El decodificador de instrucciones y la lectura del banco de registros están ahora en la etapa 2, la unidad funcional y la lectura y escritura de la memoria de datos están en la etapa 3 y la escritura en el banco de registros está en la etapa 4. Estas etapas se han etiquetado con las abreviaturas adecuadas. En la figura hemos añadido registros de *pipeline* entre las etapas, según se necesitan para pasar la información decodificada de las instrucciones a través del *pipeline* junto con el dato que está siendo procesado. Estos registros adicionales sirven para pasar a lo largo del *pipeline* la información de las instrucciones, como la información de la orden pasa a lo largo de la estación de lavado.

La primera etapa añadida es la etapa de acceso a las instrucciones, llamada IF, que está situada completamente en el control. En esta etapa, la instrucción se extrae de la memoria de instrucciones y se actualiza el valor del *PC*. Dada la complejidad adicional que supone el manejo de saltos y bifurcaciones en el diseño del *pipeline*, la actualización de *PC* se restringe aquí a un incremento, con un tratamiento más complejo que se proporciona en la siguiente sección. Entre la primera y segunda etapa hay un registro de *pipeline* que juega el papel de registrar la instrucción, denominado *IR*.

En la segunda etapa, DOF, para decodificar y acceder al operando, tiene lugar la decodificación de *IR* en señales de control. Entre las señales decodificadas, las direcciones del banco de registros AA y BA y la señal del control del multiplexor se utilizan para el acceso del operando. El resto de las señales de control decodificadas se pasan al siguiente registro de *pipeline* para utilizarlas posteriormente. Siguiendo a la etapa DOF está el segundo registro de *pipeline*, cuyos registros almacenan las señales de control que se utilizarán más tarde. La tercera etapa del *pipeline* es la etapa de ejecución, llamada EX. En esta etapa, se ejecuta una operación de la ALU, de desplazamiento o de memoria para la mayoría de las instrucciones. Así, las señales de control utilizadas en esta etapa son FS y MW. La parte de lectura de la memoria de datos *M* se considera también parte de la etapa. Para una lectura de memoria, el valor de la palabra de lectura se lee de la salida Data out de la memoria de datos. Todos los resultados generados en esta etapa, más las señales de control de la última etapa, se capturan en el tercer registro de *pipeline*.



□ **FIGURA 12-4**
Diagrama de bloques de un procesador en *pipeline*

La parte de escritura de la memoria de datos *M* se considera una parte de esta etapa, de forma que puede realizarse aquí una escritura de memoria. La información de control se mantiene al final del registro de *pipeline* compuesto por DA, MD y RW, que se usa en la etapa final de reescritura, WB.

La ubicación de los registros del *pipeline* ha balanceado el reparto de los retardos, de forma que el retardo por etapa no es mayor que 5 ns. Esto da lugar a una posible frecuencia de reloj

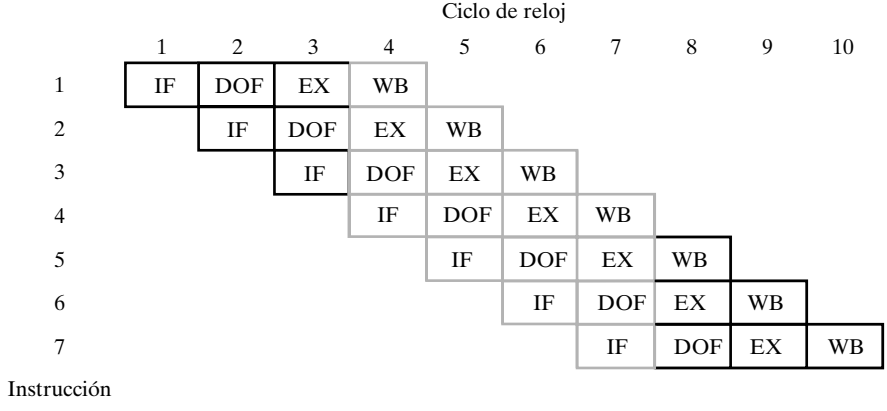
de 200 MHz, en torno a 3.4 veces la del procesador de un solo ciclo. Nótese, sin embargo, que una instrucción toma $4 \times 5 = 20$ ns para ejecutarse. Esta latencia es de 20 ns, mientras que para un procesador de un solo ciclo de reloj es de 17 ns. Sólo si se ejecutan las instrucciones una a una, se ejecutarían menos instrucciones que en un procesador de un solo ciclo de reloj.

Rendimiento y realización de un *pipeline*

Si nuestra hipotética estación de lavado se amplia a cuatro etapas, habrá hasta 4 operaciones en las etapas en un tiempo dado. Entonces, por analogía, debería ser posible tener cuatro instrucciones en las etapas del *pipeline* de nuestro procesador en un mismo instante de tiempo. Consideremos un cálculo sencillo: cargar las constantes de 1 a 7 en los siete registros, *R*1 a *R*7, respectivamente. El programa para hacer esto es como sigue (el número a la izquierda es un número que identifica a la instrucción):

```
1    LDI R1, 1
2    LDI R2, 2
3    LDI R3, 3
4    LDI R4, 4
5    LDI R5, 5
6    LDI R6, 6
7    LDI R7, 7
```

Vamos a examinar la ejecución de este programa con respecto de las etapas del *pipeline* de la Figura 12-4. Para ello emplearemos el diagrama de patrones de ejecución del *pipeline* de la Figura 12-5. En el ciclo de reloj 1, la instrucción 1 está en la etapa IF del *pipeline*. En el ciclo de reloj 2, la instrucción 1 está en la etapa DOF y la instrucción 2 está en la etapa IF. En el ciclo de reloj 3, la instrucción 1 está en la etapa EX, la instrucción 2 está en la etapa DOF, y la instrucción 3 está en la etapa IF. En el ciclo de reloj 4, la instrucción 1 está en la etapa WB, la instrucción 2 está en la etapa EX, la instrucción 3 está en la etapa DOF, y la instrucción 4 está en la etapa IF. De esta forma, al final de cuarto ciclo de reloj, la instrucción 1 ha completado su



□ **FIGURA 12-5**
Ejecución del patrón en *pipeline* del programa del número de registros

ejecución, la instrucción 2 está terminada en tres cuartos, la instrucción 3 está medio terminada, y la instrucción 4 se ha completado en un cuarto. Es decir, hemos completado $1 + 3/4 + 1/2 + 1/4 = 2.5$ instrucciones en cuatro periodos de reloj, 20 ns. Podemos ver del diagrama que completa la ejecución del programa completo de las siete instrucciones que se necesitan 10 ciclos de reloj para ejecutarlo. Es decir se necesitan 50 ns, mientras que el procesador de un solo ciclo necesita 119 ns, ejecutándose, por tanto, el programa 2.4 veces más rápido.

Supongamos ahora que examinamos la ejecución del patrón del *pipeline* detenidamente. En los primeros tres ciclos de reloj no están activas todas las etapas del *pipeline* puesto que se está llenando. En los siguientes cuatro ciclos de reloj, todas las etapas del *pipeline* están activas, como se indica en azul, y, por tanto, el *pipeline* está siendo utilizado en su totalidad. En los tres últimos ciclos de reloj no están activas todas las etapas del *pipeline* puesto que se está vaciando el *pipeline*. Si queremos encontrar la mayor mejora posible del procesador en *pipeline* sobre el procesador de un solo ciclo de reloj, comparamos los dos en la situación en la que el *pipeline* está completamente utilizado. Con estos cuatro ciclos, o 20 ns, el *pipeline* ejecuta $4 \times 4 \div 4 = 4.0$ instrucciones. En el mismo tiempo, el procesador de un solo ciclo de reloj, ejecuta $20 \div 17 = 1.18$ instrucciones. Es decir, en el mejor de los casos, el procesador en *pipeline* ejecuta $4 \div 18 = 3.4$ veces más instrucciones, en un tiempo dado, que el procesador de un solo ciclo. Véase que, aunque el *pipeline* tiene cuatro etapas, el procesador en *pipeline* no es cuatro veces más rápido que el procesador de un solo ciclo ya que los retardos de este último no se pueden dividir exactamente en cuatro segmentos iguales y por los retardos añadidos por los registros del *pipeline*. También, el llenado y vaciado del *pipeline* reduce bastante la velocidad de forma que la velocidad del procesador en *pipeline* es menor que la velocidad máxima ideal de 3.4 veces la velocidad del procesador de un solo ciclo de reloj.

El estudio del procesador en *pipeline* aquí visto, junto con el procesador de un solo ciclo y el procesador multiciclo del Capítulo 10, completa nuestro examen de tres organizaciones de control de los procesadores. Tanto las rutas de datos segmentadas como los controles que hemos estudiado aquí se han simplificado y se han eliminado elementos. A continuación presentamos dos diseños de CPUs que ilustran la combinación de las características arquitecturales del conjunto de instrucciones, la ruta de datos y la unidad de control. Los diseños son *top-down*, pero reutilizan los diseños de componentes anteriores, ilustrando la influencia de la arquitectura de conjunto de instrucciones en la ruta de datos y las unidades de control, y la influencia de la ruta de datos en la unidad de control. El material hace un uso extensivo de tablas y diagramas. Aunque reutilizamos y modificamos el diseño de componentes del Capítulo 10, la información de fondo de este capítulo no se repite aquí. Sin embargo, los punteros, se dan en las primeras secciones del libro, donde se puede encontrar información detallada. Las dos CPUs que se presentan son para un RISC que utiliza una ruta de datos segmentada con una unidad de control cableada en *pipeline* y un CISC basado en el RISC, que utiliza una unidad de control auxiliar microprogramada. Estos dos diseños representan a las dos arquitecturas distintas de conjunto de instrucciones con arquitecturas que utilizan un núcleo en *pipeline* común que contribuye a la mejora del rendimiento.

12-3 PROCESADOR DE CONJUNTO REDUCIDO DE INSTRUCCIONES

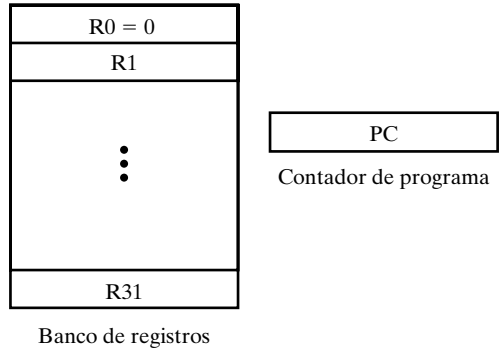
El diseño que primero vamos a examinar es un procesador de conjunto reducido de instrucciones con una ruta de datos segmentada y una unidad de control. Empezamos describiendo la arquitectura de conjunto de instrucciones RISC, que se caracteriza por los accesos a la memoria

de carga/almacenamiento, cuatro modos de direccionamiento, un único formato para las instrucciones con la misma longitud y las instrucciones que utiliza son sólo operaciones elementales. Las operaciones, parecidas a las que se realizan en un procesador de un solo ciclo, se pueden hacer mediante un solo paso a través del *pipeline*. La ruta de datos para realizar la arquitectura ISA se basa en la ruta de datos de un solo ciclo que se describió en la Figura 10-11 y se convirtió en *pipeline* en la Figura 12-2. A la hora de realizar la arquitectura RISC, se han hecho modificaciones en el banco de registros y en la unidad funcional. Estas modificaciones representan los efectos de una instrucción de longitud más larga y el deseo de incluir desplazamientos de varias posiciones entre las operaciones elementales. La unidad de control se basa en la unidad de control en *pipeline* de la Figura 12-4. Las modificaciones incluyen el soporte para instrucciones de 32 bits y una estructura más extensa del contador de programa para proceder con las bifurcaciones en el entorno del *pipeline*. En respuesta a los conflictos de datos y del control asociados con el diseño de *pipelines*, los cambios se harán tanto en el control como en la ruta de datos para mantener la ganancia en el rendimiento conseguido utilizando un *pipeline*.

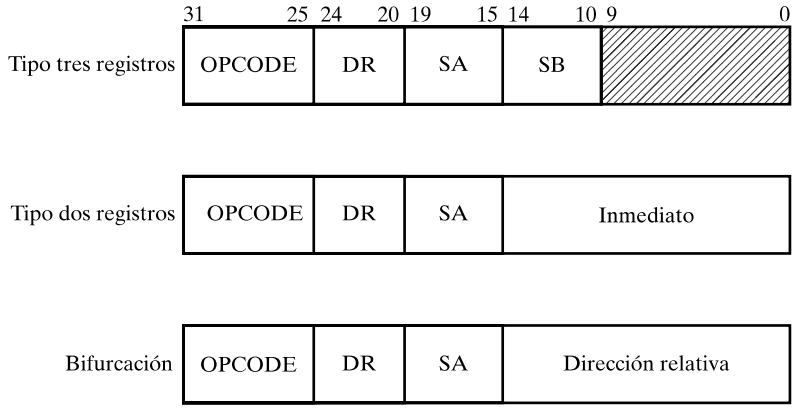
Arquitectura de conjunto de instrucciones

La Figura 12-6 muestra los registros de la CPU accesible por el programador en este RISC. Todos los registros son de 32 bits. El banco de registros tiene 32 registros, de *R*0 a *R*31. *R*0 es un registro especial que aporta el valor cero cuando se usa como fuente y descarta el resultado cuando se utiliza como destino. El tamaño del banco de registros accesible por el programador es relativamente grande en el RISC debido a la arquitectura de conjunto de instrucciones de carga/almacenamiento. Como las operaciones de manipulación de datos sólo pueden utilizar operandos de los registro, muchos de los operandos activos necesitan estar presentes en el banco de registros. De otra forma, se necesitarían muchas cargas y almacenamientos para salvar temporalmente los operandos en la memoria de datos entre las operaciones de manipulación de datos. Además, en muchos *pipelines* reales, estas cargas y almacenamientos necesitan más de un ciclo de reloj para su ejecución. Para prevenir este factor de degradación del rendimiento del RISC, se necesita un gran banco de registros.

Además del banco de registros, se proporciona un contador de programa, *PC*. Si se necesitan operaciones basadas en una pila con punteros o con un registro de status, éstas se realizan mediante una secuencia de instrucciones utilizando los registros.



□ **FIGURA 12-6**
Diagrama del conjunto de registros de la CPU de un RISC



□ **FIGURA 12-7**
Formatos de las instrucciones de la CPU RISC

La Figura 12-7 da los tres formatos de instrucciones para una CPU RISC. Los formatos utilizan una única palabra de 32 bits. Esta longitud de palabra más larga es necesaria para contener valores de dirección más reales, ya que es difícil acomodar en la CPU RISC instrucciones con palabras adicionales. El primer formato especifica tres registros. Los dos registros direccionados por los campos de registros fuentes de 5 bits SA y SB contienen los dos operandos. El tercer registro, diseccionado por el campo de registro destino de 5 bits, DR, especifica la posición del registro para guardar el resultado. Un OPCODE de 7 bits proporciona un máximo de 128 operaciones.

Los dos formatos restantes reemplazan al segundo registro con una constante de 15 bits. En el formato de dos registros, la constante actúa como un operando inmediato, y el formato de bifurcación, la constante es una *dirección relativa* (*offset*). La *dirección absoluta* es otro nombre para la dirección efectiva, particularmente si la dirección se utiliza en una instrucción de bifurcación. La dirección absoluta se forma sumando la dirección relativa al contenido del *PC*. De esta forma, la bifurcación utiliza un direccionamiento relativo basado en el valor actualizado del *PC*. Así, para bifurcarse hacia atrás de la posición actual del *PC*, el *offset*, que se utiliza como un número en complemento a 2 con extensión de signo, se suma al *PC*. La instrucción de bifurcación especifica el registro fuente SA. La ramificación o el salto se realiza si el contenido del registro fuente es cero. El campo DR se utiliza para especificar el registro en el que se almacena la dirección de retorno de la llamada al procedimiento. Finalmente, los 5 bits de la derecha de la constante de 15 bits también se usan como el número de desplazamientos SH para desplazamientos múltiples de bits.

La Tabla 12-1 contiene las 27 operaciones a realizar por las instrucciones. Se dan un mnemónico (un opcode) y una descripción de la transferencia de registros para cada operación. Todas las operaciones son elementales y se pueden describir con una sentencia sencilla de transferencia de registros. Las únicas operaciones que pueden acceder a la memoria son la Carga y el Almacenamiento. Hay un número apreciable de instrucciones inmediatas que ayudan a reducir el número de accesos a la memoria de datos y así acelerar la ejecución cuando se emplean constantes. Puesto que el campo inmediato de la instrucción es de solo 15 bits, los 17 bits más a la izquierda se deben rellenar para formar un operando de 32 bits. Además de utilizar el relleno de ceros para las operaciones lógicas, se utiliza un segundo método llamado *extensión de signo*. El bit más significativo del operando inmediato, el bit 14 de la instrucción, se toma como el bit de signo. Para formar un operando de 32 bits en complemento a 2, este bit se copia en

□ **TABLA 12-1**
Instrucciones de operación RISC

Operación	Notación simbólica	Opcode	Acción
No operación	NOP	0000000	Ninguna
Mover A	MOVA	1000000	$R[DR] \leftarrow R[SA]$
Suma	ADD	0000010	$R[DR] \leftarrow R[SA] + \text{R}[SB]$
Resta	SUB	0000101	$R[DR] \leftarrow R[SA] + \overline{R[SB]} + 1$
AND	AND	0001000	$R[DR] \leftarrow R[SA] \wedge R[SB]$
OR	OR	0001001	$R[DR] \leftarrow R[SA] \vee R[SB]$
OR Exclusiva	XOR	0001010	$R[DR] \leftarrow R[SA] \oplus R[SB]$
Complemento	NOT	0001011	$R[DR] \leftarrow \overline{R[SA]}$
Suma inmediata	ADI	0100010	$R[DR] \leftarrow R[SA] + \text{se } IM$
Resta inmediata	SBI	0100101	$R[DR] \leftarrow R[SA] + \overline{(\text{se } IM)} + 1$
AND inmediata	ANI	0101000	$R[DR] \leftarrow R[SA] \wedge (0 \parallel IM)$
OR inmediata	ORI	0101001	$R[DR] \leftarrow R[SA] \vee (0 \parallel IM)$
OR exclusiva inmediata	XRI	0101010	$R[DR] \leftarrow R[SA] \oplus (0 \parallel IM)$
Suma inmediata sin signo	AIU	1000010	$R[DR] \leftarrow R[SA] + (0 \parallel IM)$
Resta inmediata sin signo	SIU	1000101	$R[DR] \leftarrow R[SA] + \overline{(0 \parallel IM)}$
Mover B	MOVB	0001100	$R[DR] \leftarrow R[SB]$
Desplazamiento lógico a la derecha SH bits	LSR	0001101	$R[DR] \leftarrow \text{lsr } R[SA]$ SH bits
Desplazamiento lógico a la izquierda SH bits	LSL	0001110	$R[DR] \leftarrow \text{lsl } R[SA]$ SH bits
Carga	LD	0010000	$R[DR] \leftarrow M[R[SA]]$
Almacena	ST	0100000	$M[R[SA]] \leftarrow R[SB]$
Salto según registro	JMR	1110000	$PC \leftarrow R[SA]$
Pone a 1 si es menor que	SLT	1100101	If $R[SA] < \text{R}[SB]$ then $R[DR] = 1$
Bifurcación si es cero	BZ	1100000	If $R[SA] = 0$, then $PC \leftarrow PC + 1 + \text{se } IM$
Bifurcación si no es cero	BNZ	1010000	If $R[SA] \neq 0$, then $PC \leftarrow PC + 1 + \text{se } IM$
Salto	JMP	1101000	$PC \leftarrow PC + 1 + \text{se } IM$
Salto y enlaza	JML	0110000	$PC \leftarrow PC + 1 + \text{se } IM$, $R[DR] \leftarrow PC + 1$

los 17 bits. En la Tabla 12-1, a la extensión de signo del campo inmediato se le denomina IM. La misma notación, se *IM* («se» viene de *sign extension*), también representa la extensión de signo del campo dirección relativa que se estudio anteriormente.

La ausencia de las versiones almacenadas de los bits de status se maneja con el uso de tres instrucciones: Salto si es cero (*BZ, Branch if Zero*), Salto si no es cero (*Branch if Nonzero, BNZ*) y Poner a uno si es menor que (*Set if Less Than, SLT*). BZ y BNZ son instrucciones simples que determinan si un operando de un registro es cero o no y se bifurca consecuentemente. SLT almacena un valor en un registro *R*[DR] que actúa como un bit de status negativo. o Si *R*[*SA*] es menor que *R*[*SB*], se coloca un 1 en el registro *R*[*DR*]; si *R*[*SA*] es mayor que

o igual que *R*[*SB*], se coloca un cero en *R*[*DR*]. El registro *R*[*DR*] puede ser examinado mediante la consiguiente instrucción para comprobar si es cero (0) o no cero (1). De esta forma, utilizando dos instrucciones, se pueden determinar los valores relativos de los dos operandos o el signo de un operando (dejando *R*[*SB*] igual a *R*0).

La instrucción Salto y Enlaza (*Jump and Link*, JML) proporciona un mecanismo para realizar procedimientos. El valor del *PC* después de actualizarse se almacena en el registro *R*[*DR*] y luego se coloca en el *PC* la suma del *PC* y la dirección relativa de la instrucción con extensión de signo. Para el retorno de la llamada al procedimiento se puede usar la instrucción de Salto según Registro con SA igual a la DR del procedimiento de llamada. Si se llama a un procedimiento dentro de otro procedimiento, entonces cada procedimiento sucesivo que se llama necesitará su propio registro para almacenar su valor de retorno. Se puede utilizar una pila software que mueve las direcciones de retorno de *R*[*DR*] a la memoria al comienzo del procedimiento invocado y las repone en *R*[*SA*] antes del retorno.

Modos de direccionamiento

Los cuatro modos de direccionamiento en el RISC son registro, registro indirecto, inmediato y relativo. El modo se especifica con el código de operación en lugar de hacerlo en campos separados de modo. Como consecuencia, el modo para una determinada operación es fijo y no se puede variar. Las instrucciones de manipulación de datos con tres operandos utiliza el modo de direccionamiento de registro. El modo de registro indirecto, sin embargo, se aplica sólo a las instrucciones de carga y almacenamiento, la única instrucción que accede a la memoria de datos. Las instrucciones que usan el formato de dos registros tienen un valor inmediato que reemplaza al registro de dirección SB. El direccionamiento relativo se aplica exclusivamente a las instrucciones de bifurcación y de salto y así generar direcciones sólo para la memoria de instrucciones.

Cuando los programadores quieren utilizar un modo de direccionamiento no soportado por la arquitectura de conjunto de instrucciones, como el indexado, deben utilizar una secuencia de instrucciones RISC. Por ejemplo, para una dirección indexada para una operación de carga, la transferencia deseada es:

$$R15 \leftarrow M[R5 + 0 \parallel I]$$

Esta transferencia se puede llevar a cabo ejecutando dos instrucciones:

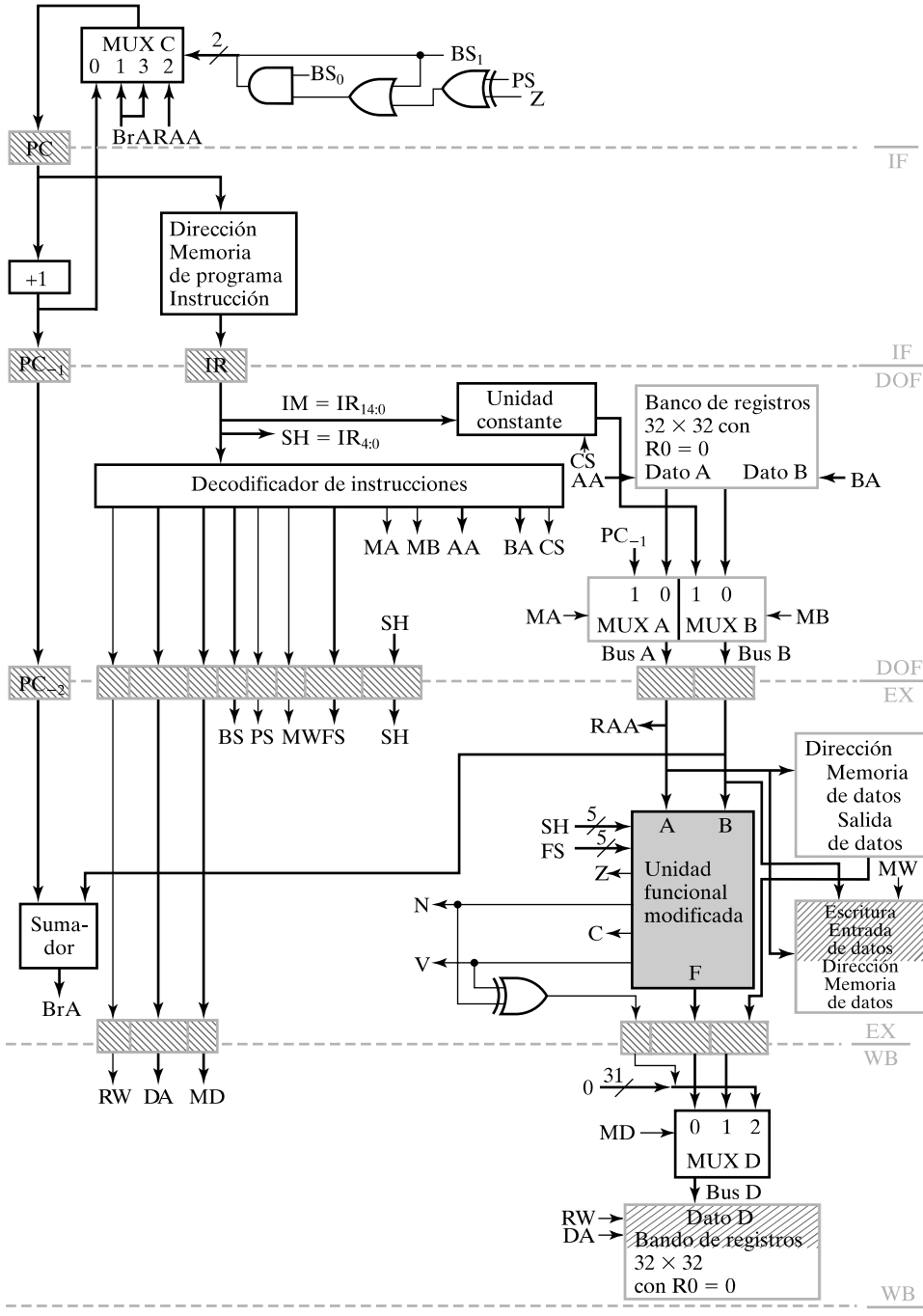
```
AIU  R9, R5, I
LD   R15, R9
```

La primera instrucción, Suma Inmediata sin Signo, forma la dirección añadiendo 17 ceros a la izquierda de *I* y suma el resultado a *R*5. La dirección efectiva resultante se almacena entonces temporalmente en *R*9. Luego, la instrucción Carga utiliza el contenido de *R*9 como dirección a la que se accede al operando y lo coloca en el registro de destino *R*5. Como por el direccionamiento indexado, *I* se toma como un offset positivo en la memoria, es adecuado usar la suma sin signo. La primera justificación para tener una suma inmediata sin signo disponible es la secuencia de operaciones para realizar los modos de direccionamiento.

Organización de la ruta de datos

La ruta de datos de la Figura 12-2 sirve aquí como base para la ruta de datos, y sólo trataremos las modificaciones. Estas modificaciones afectan al banco de registros, la unidad funcional y las

estructuras de los buses. El lector también debería remitirse a la ruta de datos de la Figura 12-2 y a la nueva ruta de datos mostrada en la Figura 12-8 para comprender completamente la siguiente discusión. Trataremos cada modificación por orden, empezando con el banco de registros.

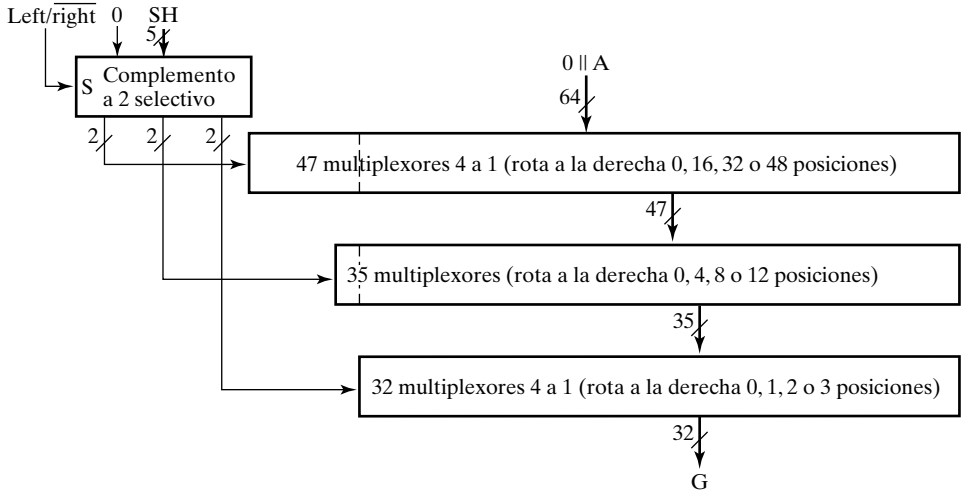


□ **FIGURA 12-8**
CPU RISC en *pipeline*

En la Figura 12-2 hay 16 registros de 16 bits y todos los registros tienen idéntica funcionalidad. En la nueva ruta de datos hay 32 registros de 32 bits. Además, la lectura del registro *R*0 da un valor constante igual a cero. Si se intenta una escritura en *R*0 se perderá el dato. Estos cambios se realizan en el nuevo banco de registro de la Figura 12-8. Todas las entradas y salidas de datos son de 32 bits. Hay 5 entradas de dirección para seleccionar los 32 registros. El valor fijo de 0 en *R*0 se realiza reemplazando los elementos de memoria de *R*0 con circuitos abiertos en las líneas que fueron sus entradas, y poniendo ceros en las líneas que fueron sus salidas.

Otra modificación importante en la ruta de datos es la sustitución del desplazador combinacional (en inglés *barrel shifter*) de una sola posición por un desplazador combinacional que permite desplazar varias posiciones. Este desplazador puede realizar el desplazamiento lógico a la derecha o a la izquierda de 0 a 31 posiciones. En la Figura 12-9 aparece el diagrama de bloques del desplazador combinacional. La entrada de datos es el operando *A* de 32 bits y la salida es el resultado *G* de 32 bits. Una señal de control, *left*/$\overline{right}$, decodificada del OPCODE, selecciona entre el desplazamiento a la izquierda o a la derecha. El campo cantidad a desplazar, $SH = IR(4:0)$ especifica el número de posiciones a desplazar el dato de entrada y toma valores entre 0 y 31. Un desplazamiento de *p* bits implica la inserción de *p* ceros en el resultado. Para proporcionar estos ceros y simplificar el diseño del desplazador, realizaremos estos desplazamientos a la derecha y a la izquierda utilizando rotaciones a la derecha. La entrada para esta rotación será la entrada de datos *A* con 32 ceros concatenados a su izquierda. Se hace un desplazamiento a la derecha rotando la entrada *p* posiciones a la derecha; se realiza el desplazamiento a la izquierda rotando $64 - p$ posiciones a la derecha. El número de posiciones se puede obtener haciendo el complemento a 2 al valor de los seis bits de $0 \,||\, SH$.

Las 63 rotaciones diferentes se pueden obtener utilizando tres niveles de multiplexores de 4 a 1, como se muestra en la Figura 12-9. El primer nivel desplaza 0, 16, 32 o 48 posiciones, el segundo nivel desplaza 0, 4, 8 o 12 posiciones y el tercer nivel desplaza en 0, 1, 2 o 3 posiciones. El número de posiciones para desplazar *A*, de 0 a 63, se puede realizar representando $0 \,||\, SH$ como un entero de tres dígitos en base 4. Desde la izquierda a la derecha, los dígitos tienen los pesos $4^2 = 16$, $4^1 = 4$ y $4^0 = 1$. Los valores de los dígitos en cada posición son 0, 1, 2 y 3. Cada dígito controla un nivel de multiplexores de 4 a 1, el dígito más significativo con-



□ **FIGURA 12-9**
Desplazador combinacional de 32 Bits

trola el primer nivel y el menos significativo el tercer nivel. Debido a la presencia de los 32 ceros en la entrada de 64 bits, se pueden usar menos de 64 multiplexores en cada nivel. Cada nivel necesita que el número de multiplexores sea 32 más el número de posiciones que sus salidas pueden ser desplazadas por los siguientes niveles. El último nivel no se puede desplazar más y por tanto necesita solamente 32 multiplexores.

La unidad funcional, la ALU se amplia a 32 bits, y el desplazador combinacional sustituye al desplazador de una posición. La unidad funcional resultante de las modificaciones utiliza los mismos códigos de función que en el Capítulo 10, excepto que los dos códigos para los desplazamientos se etiquetan ahora como desplazamientos lógicos, y algunos códigos no se utilizan. La cantidad de bits a desplazar, *SH*, es una nueva entrada de cinco bits en la unidad funcional modificada de la Figura 12-8.

Los restantes cambios de la ruta de datos se muestran en la Figura 12-8. Comenzando en la parte superior de la ruta de datos, el llenado de ceros se ha reemplazado por una Unidad Constante. La Unidad Constante realiza el llenado de ceros cuando $CS = 0$ y extensión de signo cuando $CS = 1$. MUX *A* se añade para proporcionar una ruta desde el *PC* actualizado, PC_{-1}, al banco de registros para realizar la instrucción JML.

Otro cambio en la figura ayuda a realizar la instrucción SLT. Esta lógica proporciona un 1 para que sea cargado en *R*[*DA*] si *R*[*AA*] − *R*[*BA*] < 0, y un 0 para ser cargado en *R*[*DA*] si *R*[*AA*] − *R*[*BA*] ⩾ 0. Se hace añadiendo una entrada adicional a MUX *D*. El bit 31 de la entrada es 0; el bit más a la derecha es 1 si *N* es 1 y *V* es 0 (es decir, si el resultado de la resta es negativo y no hay *overflow*). Es también 1 si *N* es 0 y *V* es 1 (es decir, si el resultado de la resta en positivo y hay *overflow*). Estos resultados representan todos los casos en los que *R*[*AA*] es mayor que *R*[*BA*] y se puede realizar utilizando una OR exclusiva de *N* y *V*.

La última diferencia en la ruta de datos es que el banco de registro no se dispara más con el flanco y no forma ya parte del registro del *pipeline* al final de la etapa de reescritura (WB). En su lugar, el banco de registro utiliza latches y se escriben mucho antes de que llegue el flanco de subida del reloj. Se proporcionan señales que permiten que el banco de registro se escriba en la primera mitad del ciclo de reloj y se lea en la última parte. A éste se le denomina banco de registro de *lectura después de escritura*, y ambos evitan la complejidad añadida en la lógica utilizada para manejar los conflictos de datos y reducir el coste del banco de registros.

Organización del control

La organización del control en el RISC de la Figura 12-4 está modificada. El decodificador de instrucciones modificado es esencial para manejar el nuevo conjunto de instrucciones. En la Figura 12-8 se ha añadido como un campo IR, también se ha puesto un campo *CS* de un bit al decodificador de instrucciones y MD se ha ampliado a dos bits. Hay, además, un nuevo registro en el *pipeline* para SH y otro de dos bits para MD.

El resto de las señales de control se han incluido para manejar la nueva lógica de control para el *PC*. Esta lógica permite cargar las direcciones en el *PC* para realizar bifurcaciones y saltos. MUX *C* selecciona entre tres fuentes diferentes para el siguiente valor del *PC*. El *PC* actualizado se usa para moverse secuencialmente a través del programa. La dirección de bifurcación *BrA* se forma a partir de la suma del valor actualizado del *PC* para la instrucción de bifurcación y la extensión de signo de la dirección relativa. El valor de *R*[*AA*] se usa como registro de salto. La selección de estos valores se controla con el campo BS. Los efectos de BS se resumen en la Tabla 12-2. Si $BS_0 = 0$, entonces se selecciona el *PC* actualizado con $BS_1 = 0$, y se selecciona *R*[*AA*] con $BS_1 = 1$. Si BS0 = 1 y BS1 = 1, se selecciona *BrA* incondicional-

□ **TABLA 12-2**
Definición de los campos de control BS y PS

Transferencia de registros	Código BS	Código PS	Comentario
$PC \leftarrow PC + 1$	00	X	Incrementa *PC*
Z: $PC \leftarrow BrA$, $\bar{Z}$: $PC \leftarrow PC + 1$	01	0	Bifurcación si cero
$\bar{Z}$: $PC \leftarrow BrA$, Z: $PC \leftarrow PC + 1$		1	Bifurcación si es distinto de cero
$PC \leftarrow R[AA]$	10	X	Salto al contenido de *R*[*AA*]
$PC \leftarrow BrA$	11	X	Bifurcación incondicional

mente. Si $BS_0 = 1$ y $BS_1 = 0$, entonces, para PS = 0 se realiza una bifurcación a *BrA* para $Z = 1$, y para PS = 1 se hace una bifurcación a *BrA* para $Z = 0$. Esto efectúa las dos instrucciones de bifurcación BZ y BNZ.

Para tener el valor del *PC* actualizado para las instrucciones de bifurcación y salto cuando se ha alcanzado la etapa de ejecución se han añadido los registros de *pipeline*, PC_{-1} y PC_{-2}. El PC_{-2} y el valor de la unidad constante son entradas para el sumador dedicado que forma *BrA* en la etapa de ejecución. Obsérvese que MUX *C* y la lógica conectada están en la etapa EX, aunque se muestra encima del *PC*. La diferencia en el ciclo de reloj relacionado provoca problemas con las instrucciones siguientes de bifurcación que manejaremos en secciones posteriores.

El corazón de la unidad de control es un decodificador de instrucciones. Este es un circuito combinacional que convierte el código de operación de *IR* en las señales de control necesarias para la ruta de datos y la unidad de control. En la Tabla 12-3, cada instrucción se identifica por su mnemónico. Para cada instrucción se da una sentencia de transferencia de registros y el opcode. Los opcodes se seleccionan de tal forma que los cuatro bits menos significativos de los siete bits coinciden con los bits del campo de control FS cuando se utilizan. Esto conduce a simplificar la decodificación. Las direcciones del banco de registros AA, BA y DA vienen directamente de SA, SB y DR, respectivamente, del *IR*.

De otra forma, para determinar los códigos de control, la CPU se parece mucho a una CPU de un solo ciclo de reloj de la Figura 10-15. Los registros de *pipeline* se pueden ignorar en esta decisión; sin embargo, es importante examinar cuidadosamente la temporización para asegurar que las diversas partes de la sentencia de transferencia de registros de la operación tenga lugar en la etapa correcta del *pipeline*. Por ejemplo, dese cuenta de que el sumador para el *PC* está en la etapa EX. Este sumador está conectado al MUX *C* y se conecta a la lógica de control y al incrementador +1 del *PC*. De esta forma, toda esta lógica está en la etapa EX y la carga del *PC*, que comienza en la etapa IF, se controla desde la etapa EX. Igualmente, la entrada *R*[*AA*] está en el mismo bloque combinacional de la lógica y no procede de la salida Dato *A* del banco de registros y si del Bus A de la etapa EX, según se muestra.

La Tabla 12-3 puede servir como base para el diseño del decodificador de instrucciones. Contiene los valores para todas las señales de control, excepto las del registro de direcciones de *IR*. Al contrario que para el decodificador de la Sección 10-8, la lógica es compleja y se debe diseñar preferiblemente con un programa de computadora de síntesis lógica.

Conflictos de datos

En la Sección 12-1 examinamos un diagrama de ejecución en *pipeline* y encontramos que llenando y vaciando el *pipeline* se reducía el *throughput* por debajo de nivel máximo alcanzable.

□ **TABLA 12-3**
Palabras de control para las instrucciones

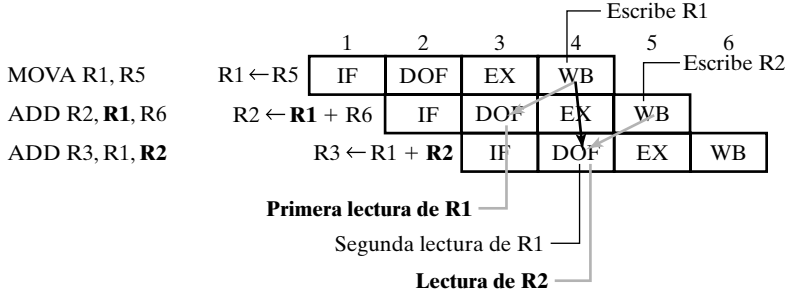
Notación simbólica	Acción	Op Code	Valores de la palabra de control								
			RW	MD	BS	PS	MW	FS	MB	MA	CS
NOP	None	0000000	0	XX	00	X	0	XXX	X	X	X
MOVA	$R[DR] \leftarrow [SA]$	1000000	1	00	00	X	0	0000	X	0	X
ADD	$R[DR] \leftarrow R[SA] + R[SB]$	0000010	1	00	00	X	0	0010	0	0	X
SUB	$R[DR] \leftarrow R[SA] + \overline{R[SB]} + 1$	0000101	1	00	00	X	0	0101	0	0	X
AND	$R[DR] \leftarrow R[SA] \wedge R[SB]$	0001000	1	00	00	X	0	1000	0	0	X
OR	$R[DR] \leftarrow R[SA] \vee R[SB]$	0001001	1	00	00	X	0	1001	0	0	X
XOR	$R[DR] \leftarrow R[SA] \oplus R[SB]$	0001010	1	00	00	X	0	1010	0	0	X
NOT	$R[DR] \leftarrow \overline{R[SA]}$	0001011	1	00	00	X	0	1011	X	0	X
ADI	$R[DR] \leftarrow R[SA] +$ se IM	0100010	1	00	00	X	0	0010	1	0	1
SBI	$R[DR] \leftarrow R[SA] +$ (se IM) + 1	0100101	1	00	00	X	0	0101	1	0	1
ANI	$R[DR] \leftarrow R[SA] \wedge$ zf IM	0101000	1	00	00	X	0	1000	1	0	0
ORI	$R[DR] \leftarrow R[SA] \vee$ zf IM	0101001	1	00	00	X	0	1001	1	0	0
XRI	$R[DR] \leftarrow R[SA] \oplus$ zf IM	0101010	1	00	00	X	0	1010	1	0	0
AIU	$R[DR] \leftarrow R[SA] +$ zf IM	1000010	1	00	00	X	0	0010	1	0	0
SIU	$R[DR] \leftarrow R[SA] + \overline{(\text{zf } IM)} + 1$	1000101	1	00	00	X	0	0101	1	0	0
MOVB	$R[DR] \leftarrow R[SB]$	0001100	1	00	00	X	0	1100	0	X	X
LSR	$R[DR] \leftarrow$ lsr $R[SA]$ SH	0001101	1	00	00	X	0	1101	X	0	X
LSL	$R[DR] \leftarrow$ lsl $R[SA]$ SH	0001110	1	00	00	X	0	1110	X	0	X
LD	$R[DR] \leftarrow M[R[SA]]$	0010000	1	01	00	X	0	XXXX	X	0	X
ST	$M[R[SA]] \leftarrow R[SB]$	0100000	0	XX	00	X	1	XXXX	X	0	X
JMR	$PC \leftarrow R[SA]$	1110000	0	XX	10	X	0	XXXX	X	0	X
SLT	If $R[SA] < R[SB]$ then $R[DR] = 1$	1100101	1	10	00	X	0	0101	0	0	X
BZ	If $R[SA] = 0$, then $PC \leftarrow PC + 1 +$ se IM	1100000	0	XX	01	0	0	0000	1	0	1
BNZ	If $R[SA] \neq 0$, then $PC \leftarrow PC + 1 +$ se IM	1010000	0	XX	01	1	0	0000	1	0	1
JMP	$PC \leftarrow PC + 1 +$ se IM	1101000	0	XX	11	X	0	XXXX	1	X	1
JML	$PC \leftarrow PC + 1 +$ se IM, $R[DR] \leftarrow PC + 1$	0110000	1	00	11	X	0	0000	1	1	1

Desafortunadamente hay otros problemas en la operación del *pipeline* que reduce el *throughput*. En ésta y en la siguiente subsección examinaremos dos de estos problemas: los conflictos de datos y el control de conflictos. Los conflictos son problemas de temporización que surgen debido a que la ejecución de una operación en un *pipeline* se retrasa en uno o más ciclos de reloj después de que se accedió a la instrucción que contenía la operación. Si la siguiente instrucción intenta utilizar el resultado de la operación como un operando antes de que el resultado esté disponible, usa el valor antiguo, dando muy probablemente un resultado erróneo. Para manejar los conflictos de datos presentamos dos soluciones, una que utiliza software y otra que usa hardware.

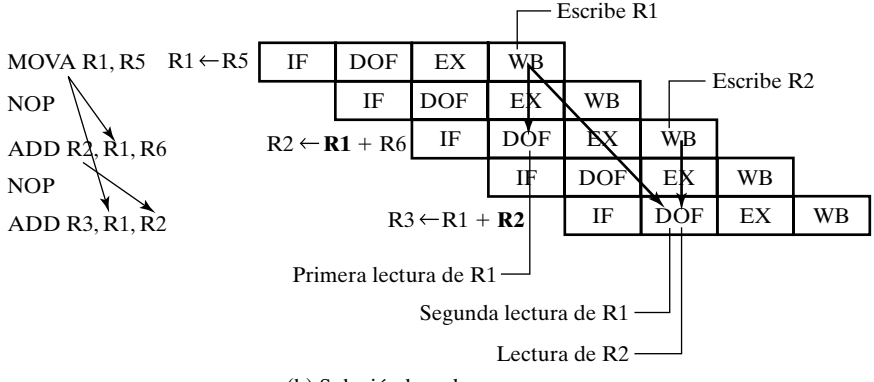
Se ilustran dos conflictos de datos examinando la ejecución del siguiente programa:

```
1   MOVA    R1, R5
2   ADD     R2, R1, R6
3   ADD     R3, R1, R2
```

El diagrama de ejecución de este programa aparece en la Figura 12-10(a). La instrucción MOVA coloca el contenido de *R*5 en *R*1 en la primera mitad de WB en el ciclo 4. Pero, como se muestra con la flecha gris, la primera instrucción ADD lee *R*1 en la última mitad de DOF en el ciclo 3, un ciclo antes de que se escriba. De esta forma, la instrucción ADD utiliza el valor antiguo que hay en *R*1. El resultado de esta operación se coloca en *R*2 en la primera mitad de WB en el ciclo 5. Sin embargo, la segunda instrucción ADD lee tanto *R*1 como *R*2 en la segunda mitad de DOF en el ciclo 4. En el caso de *R*1, el valor a leer se escribió en la primera mitad de WB en el ciclo 4. Así que la lectura del valor en la segunda mitad del ciclo 4 es el valor nuevo. La reescritura de *R*2 sucede, sin embargo, en la primera mitad del ciclo 5, después se lee por la siguiente instrucción durante el ciclo 4. Así que *R*2 no se ha actualizado con el nuevo valor en el momento que se lee. Esto da lugar a dos conflictos de datos, como se indica con la flecha grande azul de la figura. Los registros que no se han actualizado convenientemente con los valores nuevos se resaltan en azul en el programa y en la transferencia de registros de la figura. En cada uno de los casos, la lectura del registro involucrado ocurre un ciclo de reloj antes de tiempo con respecto a la escritura de este registro.



(a) Problema de conflicto de datos



(b) Solución basada en programa

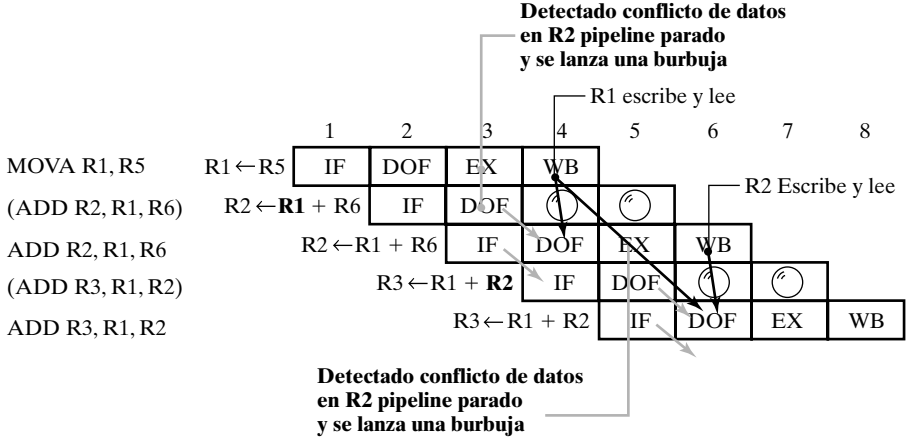
□ **FIGURA 12-10**
Ejemplo de conflicto de datos

Un posible remedio para solventar los conflictos de datos es tener un compilador o programador que genere el código de máquina para retrasar las instrucciones y así tener los valores nuevos disponibles. El programa se escribe de forma que cualquier escritura pendiente de un registro ocurre en el mismo ciclo de reloj o en uno anterior a la siguiente lectura del registro. Para conseguir esto, el programador o compilador necesita tener una información detallada de cómo opera el *pipeline*. La Figura 12-10(b) muestra una modificación del programa de tres líneas que resuelve el problema. Se insertan instrucciones de «no operación» (NOP) entre la primera y la segunda instrucción y entre la segunda y la tercera para retrasar sus respectivas lecturas, relacionadas con las escrituras, en un ciclo de reloj. El diagrama de ejecución muestra que, en el peor de los casos, este método ha escrito y ha leído apropiadamente en el mismo ciclo de reloj. Esto se indica en el diagrama con las parejas que consisten en una escritura del registro y su consiguiente lectura conectada por una flecha negra. Debido a que suponemos «una lectura después de una escritura» del banco de registros, la temporización mostrada permite que se ejecute el programa con los operandos correctos.

Este método soluciona el problema pero ¿a qué coste? Primero, el programa es, evidentemente, más largo, aunque es posible hacer otro con instrucciones que no estén relacionadas en las posiciones de las instrucciones NOP en lugar de desperdiciarlas. Además, el programa necesita dos ciclos de reloj más y reduce el *throughput* de 0.5 instrucciones por ciclo a 0.375 instrucciones por ciclo con las instrucciones NOP.

La Figura 12-11 muestra una solución alternativa que pasa por añadir hardware. En lugar de que el programador o el compilador introduzcan NOPs en el programa, el hardware es el que inserta NOPs automáticamente. Cuando se encuentra un operando en la etapa DOF que aún no se ha escrito, la ejecución asociada a la escritura se retrasa parando el flujo del pipeline en IF y en DOF durante un ciclo de reloj. Luego se reanuda el flujo terminando la instrucción cuando el operando está disponible y se accede a una nueva instrucción, como es habitual. El retraso de un ciclo de reloj es suficiente para permitir que se escriba el resultado antes de que se lea como un operando.

Cuando las acciones asociadas a una instrucción, que fluye a través del *pipeline*, se evitan que sucedan en un determinado punto, se dice que el *pipeline* contiene una burbuja en los siguientes ciclos de reloj y etapas para esa instrucción. En la Figura 12-11, cuando el flujo de la primera instrucción ADD se evita detrás de la etapa DOF, en los dos siguientes ciclos de reloj



☐ **FIGURA 12-11**
Ejemplo de una parada por conflicto de datos

pasa una burbuja a través de las etapas EX y WB, respectivamente. La retención del flujo del *pipeline* en las etapas IF y DOF retrasa durante un ciclo de reloj las microoperaciones que tienen lugar en estas etapas. En la figura se representa este retardo con dos flechas diagonales grises desde su posición inicial, en la que se evitan la realización de la microoperación, a la posición en la que se realiza la microoperación, un ciclo de reloj más tarde. Cuando se retiene el flujo en IF y DOF un ciclo de reloj extra, se dice que el *pipeline* está parado y si la causa de la parada es un conflicto de datos, a esta parada se le denomina parada por conflicto de datos.

En la Figura 12-12 se muestra una realización hardware para el manejo de conflictos de datos en un RISC segmentado. El hardware modificado o añadido se resalta con áreas sombreadas en gris. Para esta disposición en concreto de las etapas del *pipeline*, un conflicto de datos ocurrirá en una lectura del banco de registros si hay un registro de destino en la etapa de ejecución que hay que sobrescribir en el siguiente ciclo de reloj y se ha de leer en la etapa DOF en curso así como los dos operandos. Así que tenemos que determinar si tal registro existe. Esto se hace evaluando las Ecuaciones Booleanas:

$$HA = \overline{MA_{\mathrm{DOF}}} \cdot (DA_{\mathrm{EX}} = AA_{\mathrm{DOF}}) \cdot RW_{\mathrm{EX}} \cdot \sum_{i=0}^{4} (DA_{\mathrm{EX}})_i$$

$$HB = \overline{MB_{\mathrm{DOF}}} \cdot (DA_{\mathrm{EX}} = BA_{\mathrm{DOF}}) \cdot RW_{\mathrm{EX}} \cdot \sum_{i=0}^{4} (DA_{\mathrm{EX}})_i$$

y

$$DHS = HA + HB$$

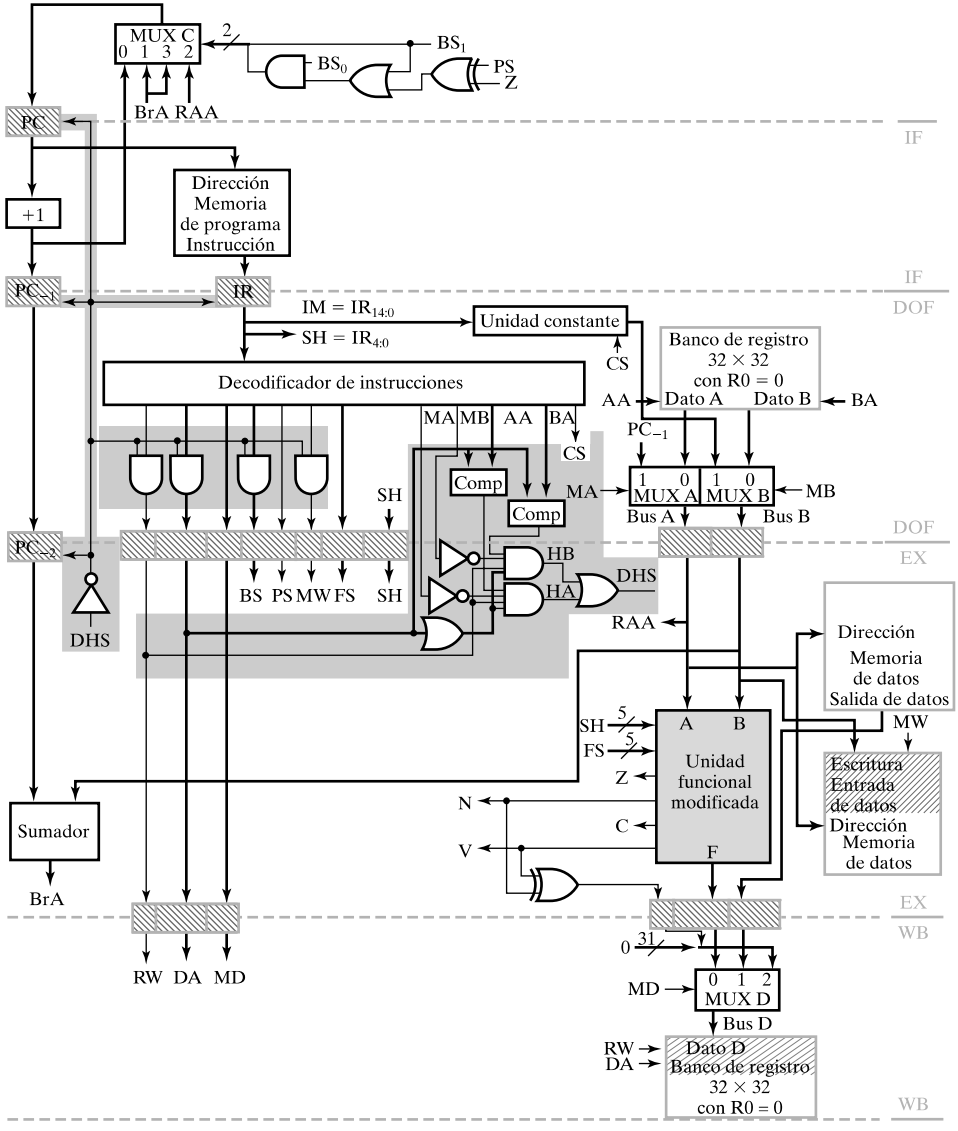
Los siguientes eventos deben ocurrir para *HA*, que representa un conflicto para el dato *A*, igual a 1:

1. *MA* en la etapa DOF debe ser 0, significando que el operando *A* procede del banco de registros.
2. *AA* en la etapa DOF es igual a *DA* en la etapa EX, queriendo decir que hay un posible registro a leer en la etapa DOF que se va a escribir en el siguiente ciclo de reloj.
3. *RW* en la etapa EX es 1, significando que el registro *DA* en la etapa EX se escribirá definitivamente en WB durante el siguiente ciclo de reloj.
4. La OR (Σ) de todos los bits de *DA* es 1, indicando que el registro que se escribirá no es *R*0 y, por tanto, es un registro que se debe escribir antes de que se lea. (*R*0 tiene siempre el valor 0 independientemente de cualquier escritura en él).

Si todas estas condiciones se cumplen es que hay una escritura pendiente para un registro en el siguiente ciclo de reloj, que es el mismo registro que está siendo leído y utilizado en el Bus *A*. Así que existe un conflicto de datos para el operando *A* del banco de registros. *HB* representa la misma combinación de eventos para el dato *B*. Si los términos *HA* o *HB* son iguales a 1, es que existe un conflicto de datos y *DHS* es 1, indicando que hace falta una parada por conflicto de datos.

La lógica que efectúa las anteriores ecuaciones se muestra sombreada en la zona sombreada en el centro de la Figura 12-12. Los bloques marcados como «Comp» son comparadores de igualdad que ponen su salida a 1 si y sólo si las entradas de 5 bits son iguales. La puerta OR a la que están conectados los cinco bits de *DA* pone un 1 en su salida cuando *DA* es distinto de 00000 (*R*0).

DHS se invierte y la señal invertida se usa para iniciar una burbuja en el *pipeline* para la instrucción que actualmente está en *IR*, así como parar el *PC* e *IR* para evitar que cambien. La burbuja, que evita que las acciones ocurran según la instrucción pasa a través de las etapas EX



□ **FIGURA 12-12**
RISC en *pipeline*: parada por conflicto de datos

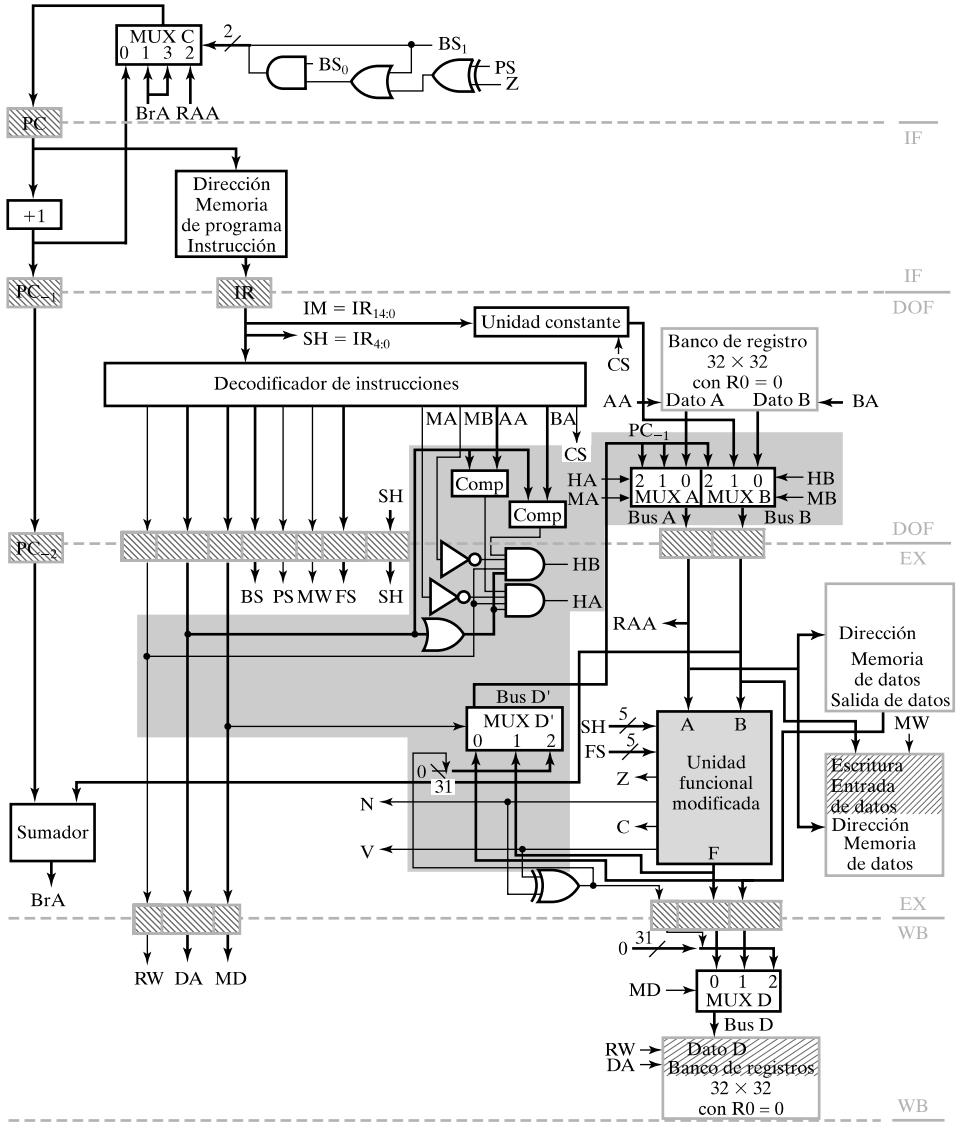
y WB, se genera utilizando puertas AND para forzar *RW* y *MW* a 0. Estos 0 evitan a la instrucción de la escritura en el banco de registros y en la memoria. Las puertas AND fuerzan también *BS* a 0 haciendo que el *PC* se incremente en lugar de cargarse durante la etapa EX para salto a registro o con una instrucción de bifurcación afectados por un conflicto de datos. Finalmente, para evitar parar el dato y que continúe en los siguientes ciclos de reloj, las puertas AND fuerzan *DA* a 0 de forma que parece que se ha escrito en *R*0, dando una condición que no produce parada. Los registros que permanecen sin cambiar en la parada son *PC*, PC_{-1}, PC_{-2} e *IR*. Estos registro se sustituyen por registros con señales de control de carga manejados por $\overline{DHS}$. Cuando $\overline{DHS}$ es 0, solicitando una parada, las señales de carga se ponen a 0 y los registros del *pipeline* mantienen sus contenidos sin cambiar en el siguiente ciclo de reloj.

Volviendo a la Figura 12-12, vemos que, en el ciclo 3, se detecta el conflicto de datos de *R*1, de forma que $\overline{DHS}$ pasa a 0 antes del siguiente ciclo de reloj. *RW*, *MW*, *BS* y *DA* se ponen a 0 y, en el flanco de reloj, se lanza una burbuja a la etapa EX para la instrucción ADD. En el mismo flanco de reloj, las etapas IF y DOF se paran y así, la información en éstas se asocian ahora al ciclo de reloj 4 en lugar de al 3. En el ciclo de reloj 4, puesto que DA_{EX} es 0, no hay parada, así que la ejecución de la instrucción ADD parada prosigue. La misma secuencia de eventos ocurre para la siguiente instrucción ADD. Nótese que el diagrama de ejecución es idéntico al de la Figura 12-10(b), excepto que las instrucciones NOP se han sustituido por instrucciones de parada, mostradas entre paréntesis. De esta forma, aunque se evita la necesidad de programar NOPs en el software, la solución de parada por conflictos de datos tiene la misma penalización en el *throughput* como el programa con NOPs.

Una segunda solución hardware, *anticipación de datos*, no tiene esta penalización. La anticipación de datos se basa en las respuestas a las siguientes cuestiones: ¿cuándo se detecta un conflicto de datos? ¿Está disponible el resultado en algún otro sitio del *pipeline* y se puede utilizar inmediatamente en la operación que tiene el conflicto de datos? La respuesta es «casi». El resultado estará en el Bus *D* pero no está disponible hasta el siguiente ciclo de reloj. El resultado se escribe en el registro de destino durante aquel ciclo de reloj. Sin embargo, la información necesaria para formar el resultado está disponible en las entradas del registro del *pipeline* que proporcionan las entradas al MUX *D*. Todo lo que se necesita para formar el resultado durante el ciclo de reloj en curso es un multiplexor que seleccione de entre tres valores, justo como lo hace MUX *D*. Se añade MUX *D*′ para generar el resultado en el Bus *D*‘. En la Figura 12-13, en lugar de leer el operando del banco de registros utilizamos la anticipación de datos para sustituir el operando por el valor de Bus *D*′. Esta sustitución se lleva a cabo mediante una entrada adicional a MUX *A* y a MUX *B* desde el Bus *D*′, según se muestra. Esencialmente, se utiliza la misma lógica que antes para detectar el conflicto de datos, excepto que la detección con las señales *HA* y *HB* se usan directamente para el dato *A* y el dato *B*, respectivamente, así que la sustitución se efectúa para el operando que tiene el conflicto de datos.

El diagrama de ejecución para la anticipación de datos para el ejemplo de tres instrucciones aparece en la Figura 12-14. El conflicto de datos para *R*1 se detecta en el ciclo 3. Esto provoca que el valor vaya a *R*1 en el siguiente ciclo, para anticiparlo de la etapa EX de la primera instrucción en el ciclo 3. El valor correcto de *R*1 entra en el registro del *pipeline* de DOF/EX en el siguiente flanco de reloj, de forma que la ejecución de la primera instrucción ADD puede proceder normalmente. El conflicto de datos en *R*2 se detecta en el ciclo 4, y el valor correcto se anticipa de la etapa EX de la segunda instrucción en este ciclo. Esto da el valor correcto a los registro del *pipeline* DOF/EX necesario para que la segunda instrucción ADD proceda normalmente. En comparación con el método de parada de datos en conflicto, la anticipación de datos no incrementa el número de ciclos de reloj necesarios para ejecutar el programa y, por tanto, no afecta al *throughput* en términos del número de ciclos de reloj que son necesarios. Sin embargo, añade un retardo combinacional, provocando que el periodo de reloj sea algo más largo.

Los conflictos de datos pueden ocurrir también en los accesos a memoria, como con los accesos a registros. No es probable que, para las instrucciones ST y LD, se pueda realizar una lectura de un dato después de una escritura en un solo ciclo de reloj. Además, algunas lecturas de la memoria pueden llevar más que un ciclo de reloj, en comparación con lo que hemos supuesto aquí. De esta forma se puede incrementar la reducción del *throughput* en un conflicto de datos, debido a un retardo más largo antes de que el dato esté disponible.

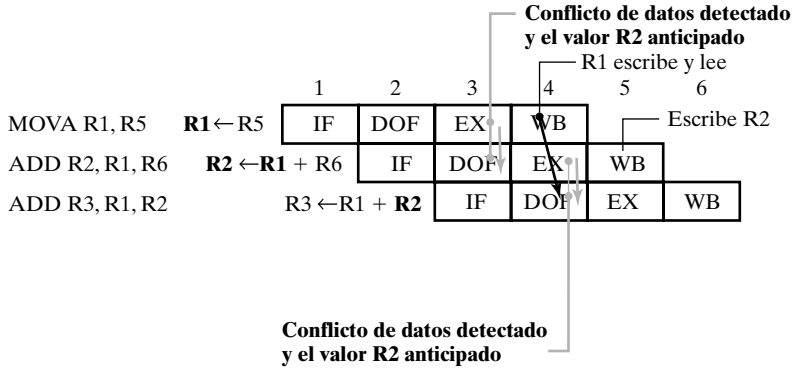


□ **FIGURA 12-13**
RISC en *pipeline*: anticipación de datos.

Control de conflictos

El control de conflictos se asocia con las bifurcaciones en el control del flujo del programa. El siguiente programa contiene una bifurcación condicional que ilustra el control de conflictos:

```
1    BZ      R1, 18
2    MOVA    R2, R3
3    MOVA    R1, R2
4    MOVA    R4, R2
20   MOVA    R5, R6
```



□ **FIGURA 12-14**
Ejemplo de anticipación de datos

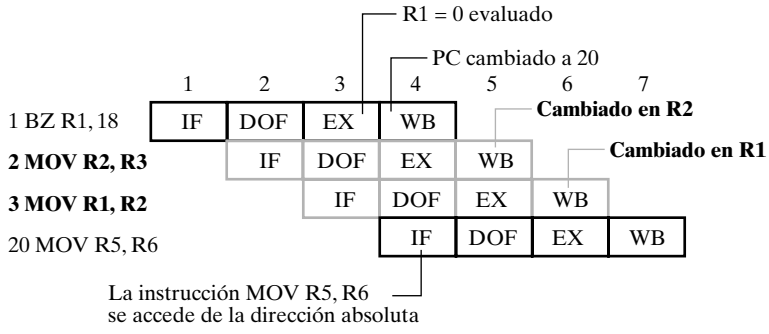
En la Figura 12-15(a) se da el diagrama de ejecución de este programa. Si *R*1 es cero, se produce una bifurcación a la instrucción de la posición 20 (recuerde que el direccionamiento es relativo al *PC*), saltando las instrucciones de las posiciones 2 y 3. Si *R*1 es distinto de cero, se ejecutan la secuencia de instrucciones de la posición 2 y 3. Supongamos que la bifurcación se toma en la posición 20 debido a que *R*1 es igual a cero. El hecho es que *R*1 es igual a 0 no se detecta hasta la etapa EX en el ciclo 3 de la primera instrucción de la Figura 12-15(a). Así que el PC se actualiza con el valor 20 en el flanco de reloj al final del ciclo 3. Pero la instrucción MOVA de las posiciones 2 y 3 están en las etapas EX y DOF, respectivamente, después del flanco de reloj. Así que, si no se toma una acción correctora, esta instrucción se terminará de ejecutar, incluso aunque la intención de programador fuese saltarlas. Esta es una forma de un *control de conflictos.*

Las instrucciones NOP se pueden utilizar para manejar el control de conflictos como se utilizó en el conflicto de datos. La inserción de NOPs se hace por el programador o por el compilador generando el programa en código máquina. El programa debe escribirse de forma que sólo las operaciones previstas, independientemente de si se realiza la bifurcación o no, se introduzcan en el *pipeline* antes de que realmente ocurra la ejecución de la bifurcación. La Figura 12-15(b) ilustra una modificación del programa de tres líneas que satisface esta condición. Se insertan dos NOPs después de la instrucción BZ. Estos dos NOPs se pueden ejecutar independientemente de si se realiza la bifurcación en la etapa EX de BZ en el ciclo 3 sin efectos adversos en la corrección del programa. Cuando el control de conflictos de la CPU se maneja de esta manera por programación, el conflicto de la bifurcación se solventa con NOPs, se le llama *bifurcación retardada*. En esta CPU se retrasa la ejecución de la bifurcación dos ciclos de reloj.

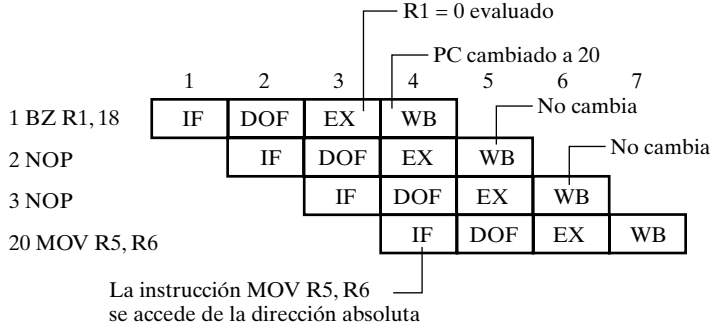
La solución con NOPs de la Figura 12-15(b) incrementa el tiempo necesario para procesar este sencillo programa en dos ciclos de reloj, independientemente de si se ejecuta la bifurcación o no. Sin embargo, aprecie que estos ciclos desperdiciados pueden evitarse a veces reorganizando el orden de las instrucciones. Suponga que se ejecutan estas instrucciones independientemente de si se lleva a cabo la bifurcación en las dos posiciones que siguen a la instrucción de bifurcación. En esta situación, la pérdida en el *throughput* se recupera completamente.

Justamente como en el caso de conflictos de datos, se puede utilizar una parada para manejar el control de conflictos. Pero también, como en el caso de conflictos de datos, la reducción del *throughput* será el mismo que con la inserción de NOPs. A esta solución se le denomina *parada de conflicto de bifurcación* y no se presentará aquí.

Una segunda solución hardware es utilizar *predicción de salto*. En su forma más simple, este método predice qué bifurcaciones no se tomarán nunca. De esta forma, se accederán a las ins-



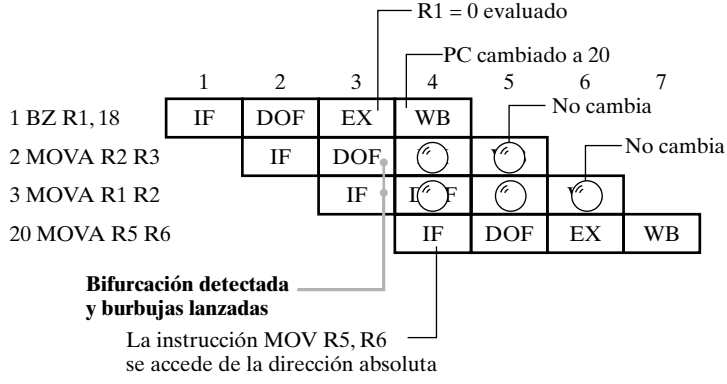
(a) Problema de conflictos en la bifurcación



(b) Solución basada en programa

□ **FIGURA 12-15**
Ejemplo de control de conflictos

trucciones y decodificarán, y se accederán a los operandos en base a la suma de 1 al valor del *PC*. Estas acciones ocurren hasta que se conoce si la bifurcación en cuestión se tomará durante el ciclo de ejecución. Si la ejecución no se efectúa, las instrucciones que ya están en el *pipeline* se pueden ejecutar debido a la predicción. Si la bifurcación se efectúa, han de cancelarse las instrucciones que siguen a la instrucción de bifurcación. Habitualmente se hace la cancelación insertando burbujas en las etapas de ejecución y reescritura de estas instrucciones. Esto se ilustra en el programa de 4 líneas de la Figura 12-16. En base a la predicción de que la bifurcación



□ **FIGURA 12-16**
Ejemplo de predicción de bifurcaciión cuando ésta se efectúa

no se realizará, se accede a las dos instrucciones MOVA después de BZ, la primera se decodifica y se accede a sus operando. Estas acciones tienen lugar en los ciclos 2 y 3. En el ciclo 3, la condición sobre la que se basa la bifurcación ha sido evaluada y se encuentra que $R1 = 0$. De esta forma, se realiza la bifurcación. Al final del ciclo 3, el *PC* se cambia a 20 y el acceso a la instrucción en el ciclo 4 se ejecuta usando el valor nuevo del *PC*. En el ciclo 3, el hecho de que se proceda con la bifurcación se ha detectado y se insertan burbujas en el *pipeline* en las instrucciones 2 y 3. Siguiendo a través del *pipeline*, estas burbujas tienen el mismo efecto que las dos instrucciones NOP. Sin embargo, como las instrucciones no están presentes en el programa, no hay retardo o penalización en el rendimiento cuando no se realiza la bifurcación.

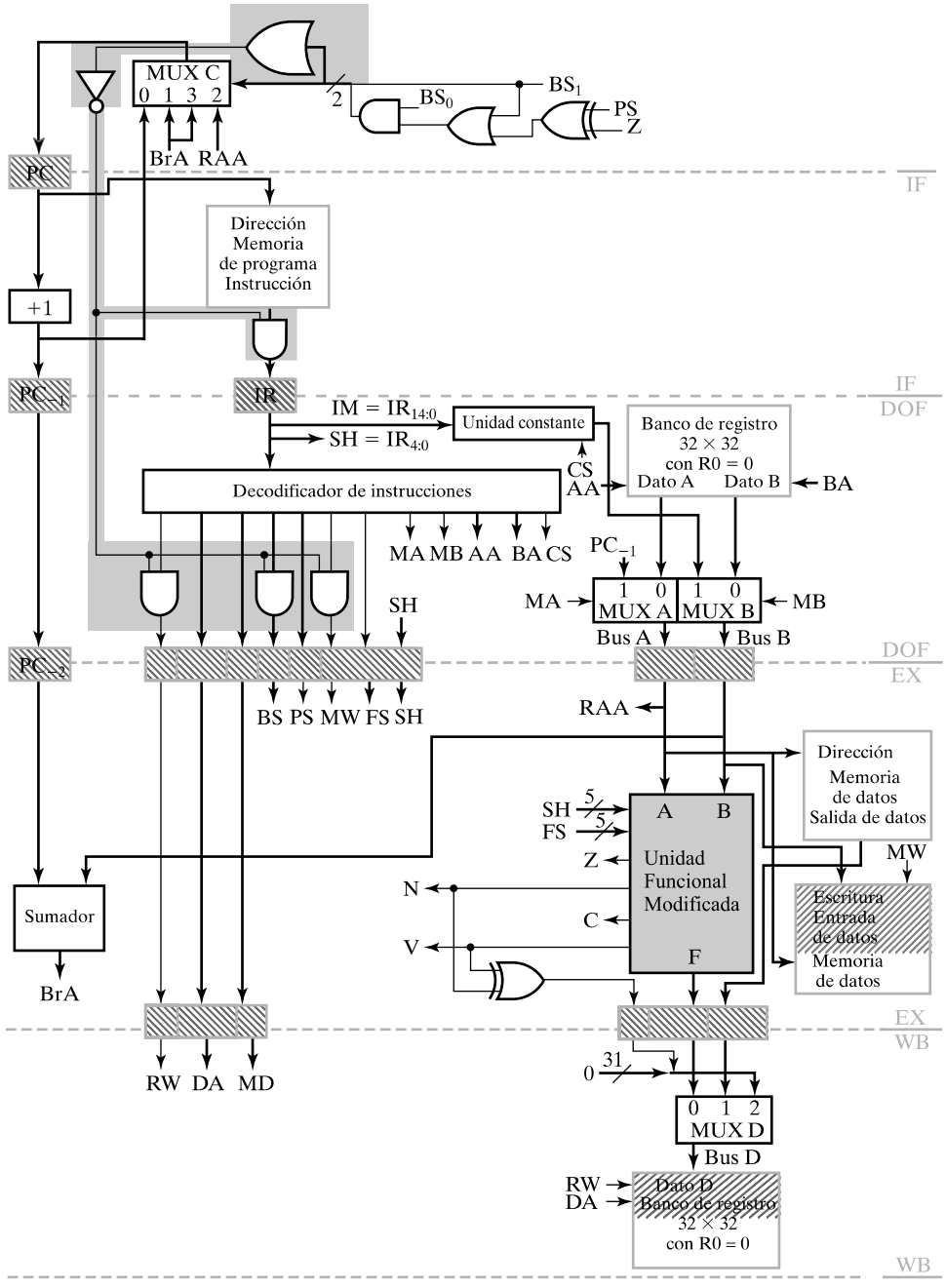
En la Figura 12-17 se muestra el hardware de predicción de bifurcación. Se determina si se efectúa una bifurcación mirando en los valores de selección a las entradas de MUX *C*. Si las dos entradas tienen el valor 01, entonces se hace la bifurcación condicional. Si las entradas valen 10, se efectúa un JMR incondicional. Si valen 11, entonces tiene lugar un JMP incondicional o JML. Por otra parte, si las entradas valen 00 significa que no se va a realizar la bifurcación. Es decir, se realiza una bifurcación para todas las combinaciones de las entradas distintas de 00 (hay al menos un 1). Lógicamente, esto se corresponde a la OR de las entradas, como se muestra en la figura. La salida de la OR se invierte y luego se hace el producto lógico, AND, con los campos *RW* y *MW*, de forma que las instrucciones siguientes a la instrucción de salto no puedan escribir en el banco de registros ni en la memoria si se toma la bifurcación. La salida invertida y el campo *BS* van a una puerta AND, de manera que no se ejecuta una bifurcación en la siguiente instrucción. Para cancelar la segunda instrucción que sigue a la bifurcación, la salida invertida de la OR va a otras puertas AND junto con las salidas de *IR*. Esto da lugar a una instrucción con todos sus bits a 0, para la que se ha definido un OPCODE correspondiente a la instrucción NOP. Sin embargo, si no se toma la bifurcación, la salida invertida de la OR es 1 y el *IR* y los tres campos de control permanecen sin cambiar, dando lugar a una ejecución normal de las dos instrucciones que siguen a la bifurcación.

La predicción de bifurcación también se puede hacer bajo la suposición de que se efectúa la bifurcación. En este caso, se debe acceder a las instrucciones y a los operandos a la ruta de la bifurcación que se está tratando. De esta forma, se debe calcular la dirección de bifurcación absoluta y usarse para acceder a la instrucción de la posición absoluta de la bifurcación. Sin embargo, en el caso de que la bifurcación no tuviese lugar, debe salvarse el valor actualizado del *PC*. Como consecuencia, esta solución necesita un hardware adicional para calcular y almacenar la dirección de bifurcación final. No obstante, si es más probable que la bifurcación se efectúe que no, la predicción de «bifurcación realizada» puede proporcionar una función de coste-rendimiento más favorable que la predicción de «bifurcación no realizada».

Por simplicidad de la presentación, hemos tratado las soluciones hardware para el manejo de conflictos al mismo tiempo. En una CPU real, estas soluciones necesitan estar combinadas. Además, puede ser necesario controlar otros conflictos, como los asociados a las escrituras y lecturas de posiciones de memoria.

12-4 PROCESADORES DE CONJUNTO DE INSTRUCCIONES COMPLEJO

Las arquitecturas de conjunto de instrucciones CISC se caracterizan mediante instrucciones complejas que son, en el peor de los casos, imposibles, y en el mejor, difíciles de realizar utilizando un procesador de un solo ciclo o a través de un *pipeline* de un solo paso. Una ISA CISC emplea con frecuencia un número determinado de modos de direccionamiento. Además, la ISA



□ **FIGURA 12-17**
RISC en *pipeline*: predicción de bifurcación

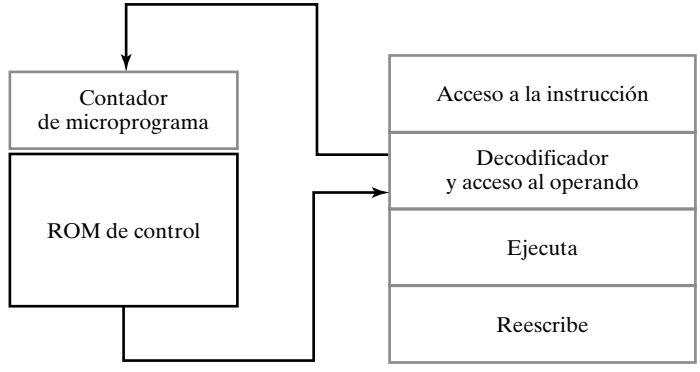
utiliza habitualmente instrucciones de longitud variable. El soporte para hacer una decisión vía bifurcación condicional es más sofisticado que el simple concepto de bifurcar sobre el contenido de cero de un registro y poner un bit de un registro a 1 basándose en la comparación de dos registros. En esta sección, se desarrolla una arquitectura básica para un CISC, con el alto rendi-

miento de un RISC para instrucciones simples, y con la mayor parte de las características de una ISA CISC como la descrita.

Supongamos que tenemos que realizar una arquitectura CISC, pero que estamos interesados en acercarnos a un *throughput* de una instrucción por ciclo de reloj pequeño de un RISC para instrucciones simples y usadas frecuentemente. Para cumplir este objetivo, usamos una ruta de datos segmentada y una combinación de un control en *pipeline* y microprogramado, como se muestra en la Figura 12-18. Se accede a una instrucción que va al IR y entra en la etapa de decodificación y acceso del operando. Si es una instrucción simple que se ejecuta completamente en solo paso a través de un RISC normal en *pipeline*, se decodifica y se accede a los operandos como es habitual. Por otro lado, si la instrucción necesita varias microoperaciones o varios accesos a la memoria secuencialmente, la etapa de decodificación genera una dirección de microcódigo, para la ROM con el microcódigo, y sustituye a las salidas habituales del decodificador con los valores de control de la ROM del microcódigo. La ejecución de las microinstrucciones de la ROM, seleccionadas por el contador del microprograma, continua hasta que se completa la ejecución de la instrucción.

Recuerde que para ejecutar una secuencia de microinstrucciones se necesita normalmente tener registros temporales en los que almacenar información. Una organización de este tipo tendrá, frecuentemente, registros temporales con el mecanismo necesario para cambiar entre los registros temporales y los recursos de almacenamiento (registros) accesible por el programador. La anterior organización soporta una arquitectura que combina las propiedades del CISC y del RISC. Se muestra que los *pipelines* y los microprogramas pueden ser compatibles y no necesitan ser vistos como mutuamente excluyentes. El uso más frecuente de dicha arquitectura combinada permite que el software diseñado existente para un CISC tome las ventajas de una arquitectura RISC mientras que se preserva la ISA existente. La arquitectura CISC-RISC es una combinación de conceptos procedentes del procesador multiciclo del Capítulo 10, de la CPU RISC de la anterior sección, y de los conceptos de microprogramación, introducidos brevemente en el Capítulo 10. Esta combinación de conceptos tiene sentido, ya que la CPU CISC ejecuta las instrucciones utilizando varios pasos a través de la ruta de datos segmentada del RISC. Para secuenciar la ejecución de las instrucciones en varios pasos, se necesita un control secuencial de considerable complejidad, de forma que se elige el control microprogramado.

El desarrollo de la arquitectura comienza con algunas modificaciones menores de la ISA RISC para obtener las capacidades deseables en la ISA CISC. A continuación, se modifica la ruta de datos para soportar los cambios de la ISA. Esto incluye modificaciones en la Unidad de



□ **FIGURA 12-18**
Organización combinada CISC-RISC

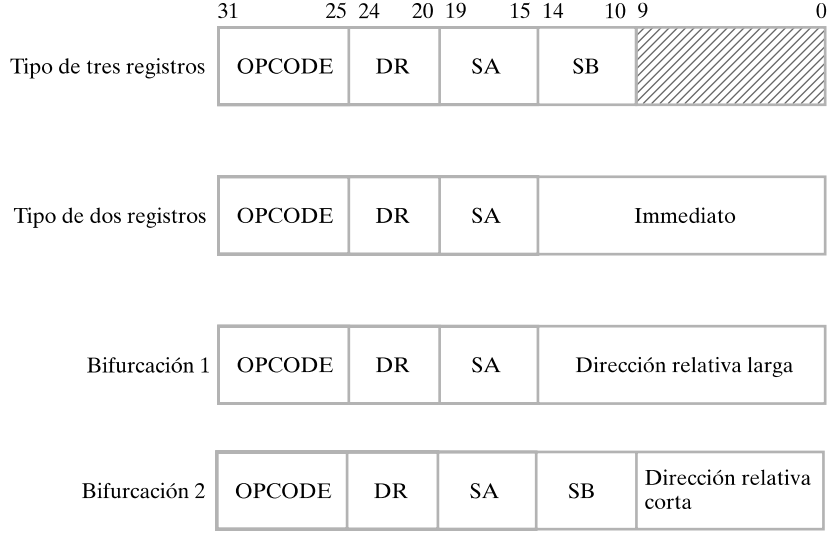
Constantes, añadir un registro de Códigos de Condiciones, *CC*, y la eliminación del hardware que soporta la instrucción SLT. Además, se modifica la lógica de direccionamiento de los 16 registros temporales para el uso de los diversos pasos de la ruta de datos con los 16 registros restantes de los recursos de almacenamiento. Esto es en comparación con los 32 registros de los recursos de almacenamiento del RISC. El siguiente paso es adaptar el control del RISC para trabajar con el control microprogramado para realizar las instrucciones que necesitan varios pasos. Para terminar, se diseña el propio control microprogramado y su operación se ilustra mediante la realización de tres instrucciones que caracterizan a la ISA CISC.

Modificaciones de la ISA

La primera modificación de la ISA RISC es añadir un nuevo formato para las instrucciones de bifurcación. En términos de las instrucciones proporcionadas en el CISC, es deseable tener la capacidad de comparar los contenidos de los dos registros fuente y de bifurcación, indicando la relación entre el contenido de los dos registros. Para realizar dicha comparación, es necesario un formato con dos campos de registros fuente, SA y SB y dirección relativa. Refiriéndonos a la Figura 12-7, el añadir el campo SB al formato de la bifurcación reduce la longitud de la longitud de la dirección relativa de 15 a 10 bits. El formato resultante de bifurcación 2, añadido para las instrucciones CISC, se muestra en la Figura 12-19.

La segunda modificación es la partición del banco de registro para proporcionar un direccionamiento de los 16 registros temporales para el uso en varios pasos de la ruta de datos. Con la partición quedan solamente 16 registros en los recursos de almacenamiento. En lugar de modificar todos los campos de dirección de registro en los formatos de las instrucciones, simplemente ignoraremos el bit más significativo de estos campos. Por ejemplo, sólo se usarán los cuatro bits más a la derecha del campo DR. Se ignorará DR_4.

La tercera modificación de la ISA RISC es añadir los códigos de condición (también llamados *flags*) como se estudió en el Capítulo 11. Los códigos de condición proporcionados se



□ **FIGURA 12-19**
Formato de las instrucciones de la CPU CISC

diseñan específicamente para utilizarlos en combinación con la bifurcación sobre la condición de cero o distinto de cero para realizar estas instrucciones y que proporcionarán un amplio espectro de decisiones, como mayor que, menor que, menor o igual que, etc. Los códigos son cero (*Z*), negativo (*N*), acarreo (*C*), overflow (*V*) y *L* (menor que). Los cuatro primeros son versiones almacenadas de las salidas de status de la unidad funcional. El bit menor que (*L*) es la OR exclusiva de *Z* y *V*, útil para realizar algunas decisiones. La inclusión del bit *L* en los códigos de condición elimina la necesidad de tener la instrucción SLT.

Para hacer un uso más efectivo de estos códigos de condición, es útil controlar si se han modificado o no en la ejecución de una microoperación concreta de una instrucción. El examen de los códigos de las instrucciones RISC de la Tabla 12-1 muestra que el bit 4 (el tercero a partir de la izquierda) del código de operación es 0 para las operaciones que hay bajo la instrucción LSL. Este bit se puede utilizar en estas instrucciones para controlar si los códigos de condición se han visto afectados por la instrucción. Si el bit es 1, el valor de los códigos de condición ha sido afectado por la ejecución de la instrucción. Si es 0, entonces los códigos de condición no han sido afectados. Esto permite un uso flexible de los códigos de condición en la toma de decisiones tanto a nivel de ISA como a nivel de microcódigo.

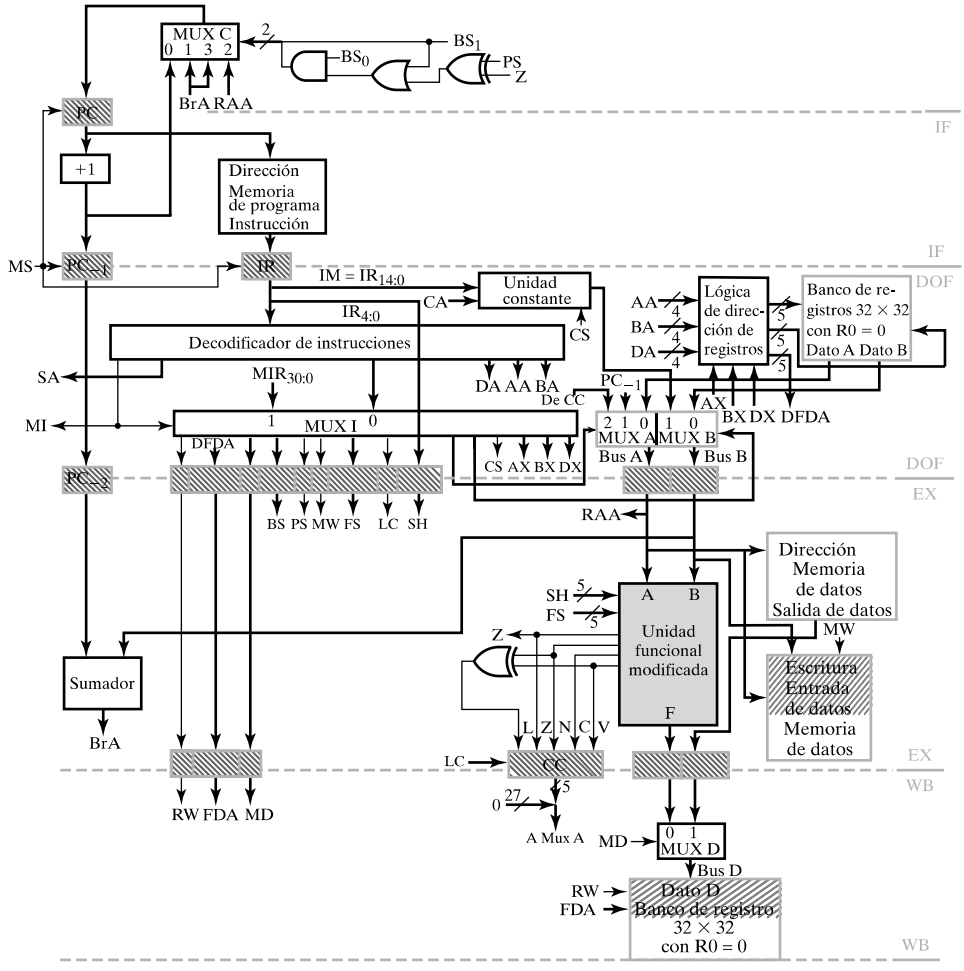
Modificaciones en la ruta de datos

Se necesitan hacer varios cambios en la ruta de datos para soportar las modificaciones de la ISA. Estos cambios se cubrirán comenzando por los componentes de la etapa DOF de la Figura 12-20.

Primero, se hacen modificaciones en la Unidad de Constantes para manejar el cambio en la longitud de la dirección relativa. La lógica añadida a la Unidad de Constantes obtiene una constante, $IM_S = IR_{9:0}$, de la constante *IM*. La extensión de signo se aplica a IM_S para obtener una palabra de 32 bits. También, para utilizar comparaciones con los valores de los códigos de condición, se proporciona una constante CA del registro de microinstrucciones, MIR, del control microprogramado. Esta constante tiene todos sus bits a cero para formar una palabra de 32 bits. El campo de control CS de la Unidad de Constantes se amplia a dos bits para realizar la selección entre 4 constantes.

Segundo, se añade la lógica del registro de direcciones del procesador multiciclo del Capítulo 10 a las entradas de direcciones del banco de registros. El propósito de este cambio es soportar la modificación de la ISA que proporciona 16 registros temporales y 16 registros que son parte de los recursos de almacenamiento. Un modo adicional soporta el uso de DX como dirección fuente para con BX como la dirección de destino del banco de registros. Esto es necesario para capturar el contenido de *R*[*DR*] para utilizarlo en los cálculos del modo de dirección de destino.

Tercero, se hacen varios cambios para permitir la modificación de añadir los códigos de condición. En la etapa DOF se añade un puerto adicional en el MUX A para proporcionar acceso a *CC*, los códigos de condición almacenados, para almacenar en los registros temporales o comparar con valores constantes. En la etapa EX, se implementa el bit del código de condición *L* (menor que) y se añade el registro de códigos de condición, *CC*, al registro del *pipeline*. La nueva señal de control LC determina si se ha cargado *CC* para la ejecución de una microoperación específica que utiliza una operación de la unidad funcional. En la etapa WB, se reemplaza la lógica que soporta la instrucción SLT por un relleno de ceros del valor de *CC*, que se pasa al nuevo puerto del MUX *A*. Como la nueva estructura de los códigos de condición proporciona el soporte para la misma decisión que hacía SLT y para más, ya no se necesita el soporte para SLT.



□ **FIGURA 12-20**
CPU CISC en *pipeline*

Modificaciones de la unidad de control

El añadir un control microprogramado a la unidad de control para soportar la realización de instrucciones que utilizan varios pasos a través del *pipeline* provoca cambios significativos del control existente, como se muestra en la Figura 12-20. El control microprogramado es una parte del hardware de decodificación de la instrucciones de la etapa DOF pero también interactúa con otras partes del control. Por conveniencia se describirán por separado.

Un vistazo rápido de la ejecución de una instrucción en varios pasos proporciona una perspectiva de los cambios de la unidad de control. El *PC* apunta a la instrucción de la memoria de instrucciones. El acceso de la instrucción se realiza en la etapa IF, y en el siguiente flanco de reloj se carga en *IR* y se actualiza el *PC*. La instrucción se identifica como una instrucción de varios pasos a partir de su opcode. La decodificación del opcode cambia la señal MI a 1 para indicar que esta instrucción es para usar el control microprogramado. El decodificador también genera una dirección de comienzo de 8 bits, SA, que identifica el comienzo del microprograma en la ROM de microcódigo. Como son necesarios varios pasos a través del *pipeline* para reali-

zar la instrucción, se debe prevenir la carga de las siguientes instrucciones en *IR* y la posterior actualización del *PC*. La señal MS, que genera la lógica del control microprogramado, se pone a 1 y detiene el *PC* y el *IR*. Esto evita que se incremente el *PC*, pero permite que *PC* + 1 continúe por el *pipeline* hasta PC_{-1} y PC_{-2} para utilizarlo en una bifurcación. Esta parada permanece hasta que la instrucción con varios pasos se haya ejecutado o hasta que haya una acción de bifurcación o de salto sobre el *PC*. Además, cuando MI = 1, la mayoría de los campos de la instrucción decodificada se reemplazan con los campos de la instrucción en curso, que es una NOP decodificada. Esta sustitución de 31 bits del campo, realizada por MUX *I*, evita que la misma instrucción provoque cualquier acción directa. Algunos cambios se han hecho en la palabra de control para controlar los recursos de la ruta de datos modificada. Los campos CS y MA se han ampliado a dos bits cada uno, y se ha añadido el campo LC. En este punto, el control microprogramado está controlando ahora el *pipeline* y proporciona una serie de microinstrucciones (palabras de control) para llevar a cabo la ejecución de la instrucción. El formato de la palabra de control es como el del procesador multiciclo e incluye campos como SH, AX, BX y DX. Se modifica DX para que cuadre con los cambios del registro de direcciones descrito para la ruta de datos. Además, el control microprogramado tiene que interactuar con la ruta de datos para realizar las decisiones. Esta interacción incluye la aplicación de una constante, CA, usa los códigos de condición, *CC*, y utiliza la señal de detección de cero *Z*.

Para soportar las operaciones que acabamos de presentar, los cambios hechos a la unidad de control son los siguientes:

1. añadir la señal de parada MS al *PC*, PC_{-1} e *IR*,
2. cambiar el decodificador de instrucciones para generar MI y SA,
3. ampliar los campos CS y MA a dos bits,
4. añadir MUX *I*, y
5. añadir los campos de control AX, BX, DX y LC.

Las definiciones de los campos nuevos y modificados se dan en la Tabla 12-4.

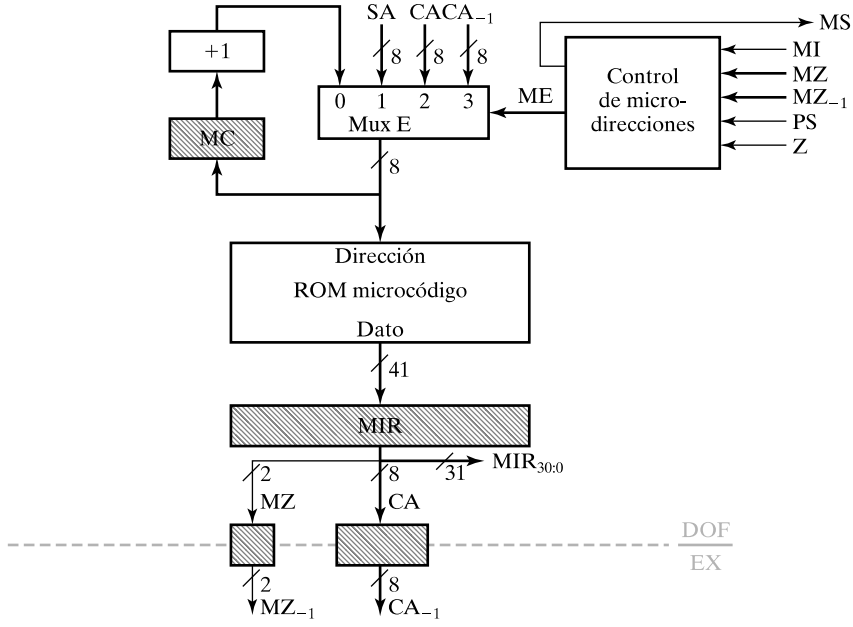
□ **TABLA 12-4**
Campos modificados o añadidos a la Palabra de Control (Microinstrucción) para el CISC

Campos de control			Campos de registro		CS		MA		LC	
MZ 2b	**CA 8h**	**BS P 2b SS**	**Acción**	**Código 5h**	**Acción**	**Código 2b**	**Acción**	**Código 2b**	**Acción**	**Código**
Ver Tabla 12-3	Siguiente dirección o constante	Ver Tabla 12-2	**AX, BX**		zf *IM*	00	Dato A	00	Retiene CC	0
					se *IM*	01	PC_{-1}	01	Carga CC	1
			R[SA], R[SB]	0X	se IM_S	10	0 || CC	10		
			R_{16}	10	zf CA	11				
			…	…						
			R_{31}	1F						
			DX							
			Fuente R[DR] y destino R[SB]	00						
			Destino R[DR] con X ≠ 0	0X						
			R_{16}	10						
			…	…						
			R_{31}	1F						

Esto completa los cambios de la unidad de control, excepto el control microprogramado añadido que se presenta en la siguiente sección.

Control microprogramado

En la Figura 12-21 aparece un diagrama de bloques del control microprogramado para las instrucciones. El control se centra en la memoria ROM con el microcódigo, que tiene direcciones de 8 bits y almacena hasta 256 microinstrucciones de 41 bits. El contador de microprograma, *MC*, almacena la dirección correspondiente a la instrucción en curso almacenada en el registro de microinstrucciones, *MIR*. Las direcciones de la memoria ROM las proporciona el MUX *E*, que selecciona a partir del *MC* incrementado la dirección de salto obtenida de la microinstrucción, CA, el valor anterior de la dirección de salto, CA_{-1}, y la dirección de comienzo a partir del decodificador de instrucciones de la unidad de control, SA. La Tabla 12.5 define los 2 bits de entrada de selección, ME, para el multiplexor MUX *E* y el bit de parada, MS, en términos del nuevo campo de control MZ más otras variables. Esta función se lleva a cabo mediante la lógica de control de las microdirecciones. Para cambiar el contexto del estudio, en la posición 0 de la memoria ROM, el estado IDLE 0 del control microprogramado contiene una instrucción que es una NOP con todos sus bits a 0. Esta microinstrucción tiene MZ = 0 y CA = 0. De la Tabla 12-5, con MI = 0, la dirección del microprograma es CA = 0, que provoca que el control



□ **FIGURA 12-21**
CPU CISC en *pipeline*: control microprogramado

□ **TABLA 12-5**
Direcciones de control entradas salidas

Inputs					Outputs			
MZ_{-1}	MZ	MI	PS	Z	ME_1	ME_0	MS	Transferencia de registro debido a ME
11	01	X	0	0	0	0	1	$\overline{PS} \cdot \overline{Z}$: $MC \leftarrow MC + 1$
11	01	X	0	1	0	1	1	$\overline{PS} \cdot Z$: $MC \leftarrow CA_{-1}$
11	01	X	1	0	0	1	1	$PS \cdot \overline{Z}$: $MC \leftarrow CA_{-1}$
11	01	X	1	1	0	0	0	$\overline{PS} \cdot Z$: $MC \leftarrow MC + 1$
0X	01	X	X	X	0	0	1	$MC \leftarrow MC + 1$
X0	01	X	X	X	0	0	1	$MC \leftarrow MC + 1$
XX	00	0	X	X	1	0	0	$MC \leftarrow CA$
XX	00	1	X	X	0	1	1	$MC \leftarrow SA$
XX	10	X	0	X	1	0	0	$\overline{PS}$: $MC \leftarrow CA$
XX	10	X	1	X	1	0	1	PS: $MC \leftarrow CA$
XX	11	X	X	X	0	0	1	$MC \leftarrow MC + 1$

permanezca en este estado hasta que MI = 1. Con MI = 1, la dirección de comienzo, SA, se aplica para acceder a la primera microinstrucción del microprograma de la instrucción compleja que está retenida en *IR*. En la unidad de control, MI = 1 también cambia MUX *I* de la palabra de control procedente del decodificador a la porción de 31 bits de *MIR*, que es una instrucción NOP. Además, la salida MS del control de microdirecciones pasa a valer 1, parando *PC*, PC_{-1} y el *IR* del control principal. En el siguiente flanco de reloj se accede a la microinstrucción a partir de que la dirección de comienzo SA entra en el *MIR*, y ahora el *pipeline* se controla con el microprograma.

En la Figura 12-21 se necesitan dos registros en el *pipeline* como parte del control microprogramado. Los valores almacenados en el pipeline, MZ_{-1} y CA_{-1}, hacen falta para la ejecución de una microbifurcación condicional ya que la comprobación del valor de *Z* se efectúa durante el ciclo de ejecución de la instrucción de microbifurcación, un ciclo de reloj después de entrar en el *MIR*.

Durante la ejecución del microprograma, la microdirección se controla mediante MZ, MZ_{-1}, MI, PS y Z. MZ_{-1} = 11, MZ = 01 ya que la microinstrucción que sigue a la microbifurcación condicional debe ser una NOP. Bajo estas condiciones, los valores de ME se controlan mediante PS y *Z* con MS = 1. Cuando PS y *Z* tienen valores opuestos, se efectúa una bifurcación condicional a la microdirección de CA_{-1}. De lo contrario, para MZ_{-1} = 11 y MZ = 01, la siguiente microdirección pasa a ser el valor incrementado de MC.

Para $MZ_{-1} \neq 11$, MZ, MI y PS controlan las microdirecciones. Para MZ = 00, los valores de *ME* y *MS* se controlan con MI. Para MI = 0, la siguiente microdirección es CA y *MS* = 0, que se corresponden con el estado de espera (IDLE) del control microprogramado. Para MI = 1, la siguiente microdirección es SA y *MS* = 1, que selecciona la siguiente microinstrucción de la memoria ROM del microcódigo y que detiene los dos primeros registros del *pipeline*. Para MZ = 01, la siguiente microdirección es el valor incrementado de *MC*, que adelanta la ejecu-

ción de la siguiente microinstrucción de la secuencia. Para MZ = 10, se realiza un salto incondicional en el control del microcódigo y el valor de MS se controla con PS. PS = 1 hace que *MS* = 1, continuando la ejecución del microprograma. PS = 0 fuerza *MS* = 0, eliminando la parada y devolviendo el control al *pipeline*. Esto provoca que MI sea 0 (si la nueva instrucción no es tampoco una compleja). Si CA = 0, el control microprogramado se queda en el estado IDLE hasta que MI = 1. Para que esto suceda, la instrucción final del microprograma debe tener MZ = 10, PS = 0 y CA = 0.

Microprograma para instrucciones complejas

Tres ejemplos ilustran la realización de instrucciones complejas utilizando las posibilidades que el CISC proporciona utilizando el diseño que se completó anteriormente. En la Tabla 12-6 se dan los microprogramas resultantes.

EJEMPLO 12-1 Instrucción LD con direccionamiento indirecto indexado (LII)

La instrucción LII suma la dirección relativa al contenido de un registro que se utiliza como registro índice. En el paso de indirección, la dirección indexada que se forma se utiliza después para acceder a la dirección efectiva de la memoria. Finalmente, la dirección efectiva se utiliza para acceder al operando de la memoria. El opcode de esta instrucción es 0110001, y la instrucción usa el formato inmediato con el campo de registro SA y una dirección relativa de 15 bits. Cuando se accede a la instrucción LII y aparece en IR, el decodificador de instrucciones pone MI a 1 y proporciona la dirección del microcódigo, que se representa simbólicamente en la Tabla 12-6 como LII0. La primera microinstrucción a ejecutar es la que aparece en la dirección IDLE. Esta microoperación ejecuta una instrucción NOP en la ruta de datos y la memoria, pero ante la presencia de MI = 1, la dirección de control selecciona SA como la siguiente dirección de la microinstrucción, dejando, por tanto, el estado IDLE. La microinstrucción forma la dirección indexada e incrementa la dirección de *MC* para acceder a la siguiente microinstrucción LII1. Esto provoca que a la microinstrucción NOP de la dirección LII1 se acceda para su ejecución en el *pipeline*. Esta NOP se ha insertado puesto que el resultado de la microinstrucción LII0 no está colocada en *R*16 hasta la etapa WB. La siguiente instrucción en LII2 accede a la dirección efectiva de la memoria. A continuación se necesita una instrucción NOP debido al retardo en el ciclo de reloj para escribir la dirección efectiva a *R*17. La microinstrucción en LII4 aplica la dirección efectiva a la memoria para obtener el operando y colocar en el registro destino *R*[DR]. Ya que esto termina la realización de LII, el control microprogramado de *MC* regresa a IDLE y se accede a la siguiente instrucción a LII de la memoria de instrucciones utilizando la dirección en el *PC*. ■

En la Tabla 12-6, se describe esta secuencia de microinstrucciones en la columna Acción mediante sentencias de transferencia de registro, y se proporcionan los nombres simbólicos para las direcciones de las microinstrucciones en la memoria ROM con el microcódigo. El resto de columnas de la tabla proporcionan la codificación de los campos de la microinstrucción. Estos códigos se han seleccionado de las Tablas 10-12, 12-2, 12-3 y 12-5 para realizar las transferencias de registro. Hay que destacar la ocurrencia en la instrucción LII4 de MC = 0, PS = 0 y CA = IDLE (00) que hace que el control del microprograma regrese al estado IDLE y el control del programa vuelve al control del *pipeline*.

□ TABLA 12-6
Ejemplo de microprogramas para la arquitectura CISC

		Microinstrucciones														
Acción	**Dirección**	**MZ**	**CA**	**RW**	**DX**	**MD**	**BS**	**PS**	**MW**	**FS**	**LC**	**MA**	**MB**	**AX**	**BX**	**CS**
Microinstrucciones compartidas																
MI: $MC \leftarrow SA$, $\overline{MI}$: $MC \leftarrow 00$	IDLE	00	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$MC \leftarrow MC + 1$ (NOP)	Arbitrario	01	XX	0	00	0	00	0	0	0	0	00	0	00	00	00
Carga Indirecta Indexada (LII)																
$R_{16} \leftarrow R[SA]$ + zf IM_L	LII0	01	00	1	10	0	00	0	0	2	0	00	1	00	00	00
$MC \leftarrow MC + 1$ (NOP)	LII1	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$R_{17} \leftarrow M[R_{16}]$	LII2	01	00	1	11	1	00	0	0	0	0	00	0	10	00	00
$MC \leftarrow MC + 1$ (NOP)	LII3	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$R[DR] \leftarrow M[R_{17}]$	LII4	10	IDLE	1	01	1	00	0	0	0	0	00	0	11	00	00
Comparación menor o igual que (BLE)																
R[SA] − R[SB], $CC \leftarrow L \,\|\| Z \,\|\| N \,\|\| C \,\|\| V$	BLE0	01	00	0	01	0	00	0	0	5	1	00	0	00	00	00
$MC \leftarrow MC + 1$ (NOP)	BLE1	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$R_{31} \leftarrow CC \wedge 11\,000$	BLE2	01	18	1	1F	0	00	0	0	8	0	10	1	00	00	11
$MC \leftarrow MC + 1$ (NOP)	BLE3	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
if $(R_{31} \neq 0)$ $MC \leftarrow BLE7$ else $MC \leftarrow MC + 1$	BLE4	11	BLE7	0	00	0	00	1	0	0	0	00	0	1F	00	00
$MC \leftarrow MC + 1$ (NOP)	BLE5	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$MC \leftarrow IDLE$	BLE6	00	IDLE	0	00	0	00	0	0	0	0	00	0	00	00	00
$PC \leftarrow (PC_{-1})$ + se IM_L, $MC \leftarrow$ IDLE	BLE7	10	IDLE	0	00	0	11	0	0	0	0	01	1	00	00	10
Mover Bloque de Memoria (MMB)																
$R_{16} \leftarrow R[SB]$	MMB0	01	00	1	10	0	00	0	0	C	0	00	0	00	00	00
$MC \leftarrow MC + 1$ (NOP)	MMB1	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$R_{16} \leftarrow R_{16} - 1$	MMB2	01	01	1	10	0	00	0	0	5	0	00	1	00	00	11
$R_{17} \leftarrow R[DR]$	MMB3	01	00	1	00	0	00	0	0	C	0	00	0	00	11	00
$R_{18} \leftarrow R[SA] + R_{16}$	MMB4	01	00	1	12	0	00	0	0	2	0	00	0	00	10	00
$R_{19} \leftarrow R_{17} + R_{16}$	MMB5	01	00	1	13	0	00	0	0	2	0	00	0	11	10	00
$R_{20} \leftarrow M[R_{18}]$	MMB6	01	00	1	14	1	00	0	0	0	0	00	0	12	00	00
$MC \leftarrow MC + 1$ (NOP)	MMB7	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$M[R_{19}] \leftarrow R_{20}$	MMB8	01	00	0	00	0	00	0	1	0	0	00	0	13	14	00
if $(R_{16} \neq 0)$ $MC \leftarrow MMB2$	MMB9	11	MMB2	0	00	0	00	1	0	0	1	00	0	10	00	00
$MC \leftarrow MC + 1$ (NOP)	MMB10	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$MC \leftarrow$ IDLE	MMB11	10	IDLE	0	00	0	00	0	0	0	0	00	0	00	00	00

EJEMPLO 12-2 Bifurcación sobre menor o igual que (BLE)

La instrucción BLE compara el contenido de los registros *R*[SA] y *R*[SB]. Si *R*[SA] es menor o igual que *R*[SB], entonces el *PC* se bifurca a *PC* + 1 más la dirección relativa corta con extensión de signo (IM_S). En otro caso se utiliza el *PC* incrementado. El opcode de la instrucción es 1100101.

Las transferencias de registro para la instrucción se dan en la columna Acción de la Tabla 12-6. En la microinstrucción BLE0, se resta *R*[SB] de *R*[SA] y se capturan los códigos de condición L hasta V en el registro *CC*. Debido al retardo de un ciclo en la escritura en *CC*, es necesaria una instrucción NOP en la microinstrucción BLE1. *R*[SA] es menor o igual que *R*[SB] si (L + Z) = 1 (en esta expresión el + es una OR). De esta forma, de los cinco bits del código de condición, sólo interesan L y Z. Así, en la instrucción BLE2, los tres bits menos significativos de *CC* se enmascaran utilizando la máscara 11 000, haciendo una operación AND con *CC*. El resultado se coloca en el registro R_{31} y, en BLE3, es necesario utilizar una instrucción NOP para esperar a que se escriba R_{31}. En BLE4 se produce una bifurcación si R_{31} es distinto de cero. Si R_{31} es distinto de cero, L + Z = 1 resultando que *R*[SA] es menor o igual que *R*[SB]. En caso contrario, tanto L como Z son cero, indicando que *R*[SA] no es menor o igual que *R*[SB]. Debido a esta microbifurcación, es necesaria una instrucción NOP en BLE5. La conexión a MUX *E* necesita una sola NOP después de la microbifurcación en lugar de las dos que se necesitarían en la bifurcación condicional en el control principal. Si la bifurcación no se realiza, se ejecuta la siguiente microinstrucción BLE6, permaneciendo *MC* en el estado IDLE y reactivando el control del *pipeline* para ejecutar la siguiente instrucción. Si se realiza la bifurcación, se ejecuta la microinstrucción BLE7, colocando en el *PC* el valor *PC* + 1 + *BrA* para acceder a la siguiente instrucción cuando la microinstrucción alcanza la etapa EX. Nótese que tal bifurcación sobre el valor del *PC* solamente puede tener lugar después de que MS sea 0 y se reactive el *pipeline*. A este respecto, existe un control de conflictos para esta instrucción en el control principal, por lo que debe seguir una NOP. Los códigos para los campos de esta microinstrucción aparecen en la Tabla 12-6. ■

EJEMPLO 12-3 Mover un bloque de memoria (MMB)

La instrucción MMB copia un bloque de información de un conjunto de posiciones contiguas de la memoria en otro. Su opcode es 0100011 y utiliza el formato del tipo de tres registros. El registro *R*[SA] especifica la dirección A, que es la dirección de comienzo del bloque fuente de la memoria, y el registro *R*[DR] especifica la dirección B, que es la dirección de comienzo del bloque de destino. *R*[SB] da el número *n* de palabras en el bloque.

Las transferencias de registro de la instrucción se dan en la columna Acción de la Tabla 12-6. En la microinstrucción MMB0 se carga *R*[SB] en R_{16}. MMB1 contiene una NOP esperando que se escriba en R_{16}. En MMB2 se decrementa R_{16}, proporcionando un índice con *n* valores, desde *n* − 1 hasta 0, para usarlo en el direccionamiento que copia las *n* palabras. Como *R*[DR] es un registro de destino, no está normalmente disponible como fuente. Pero para hacer la manipulación de las direcciones de las posiciones de destino, es necesario que su valor se coloque en un registro que actúe como fuente. Así, en MMB3, el valor de *R*[DR] se copia en el registro R_{17} utilizando el código de registro DX = 00000, que trata a *R*[DR] como fuente y al registro especificado en el campo BX, R_{17}, como destino. En las microinstrucciones MMB4 y MMB5, R_{16} se suma a *R*[SA] y a *R*[SB] para que sirvan como punteros de las direcciones de los bloques. Debido a estas operaciones, las palabras de los bloques se transfieren primero desde las posiciones

más altas. En MMB6, la primera palabra se transfiere desde la primera dirección fuente de la memoria al registro temporal R_{20}. En MMB7 aparece una NOP que permite la escritura del valor de R_{20} con MMB6 antes de que MMB8 use el valor. En MMB8, se transfiere la primera palabra desde R_{20} a la primera dirección de destino de la memoria. En MMB9 se realiza un salto sobre cero basado en el contenido de R_{16} para determinar si se han transferido las palabras del bloque. Si no es así, entonces la siguiente microdirección en la que comienza la siguiente transferencia de palabras es MM2. Si R_{16} es igual a cero, la siguiente microinstrucción es la NOP colocada en MMB10 debido a la bifurcación. La microinstrucción de MMB11 pone a *MC* en IDLE y devuelve la ejecución al control del *pipeline*.

Los códigos de las microinstrucciones aparecen en la Tabla 12-6. El código está formado por un solo registro y transferencia de memoria con una sola bifurcación para proporcionar la capacidad de realizar un bucle y NOPs para manejar los conflictos de datos y de control. ■

12-5 MÁS SOBRE DISEÑO

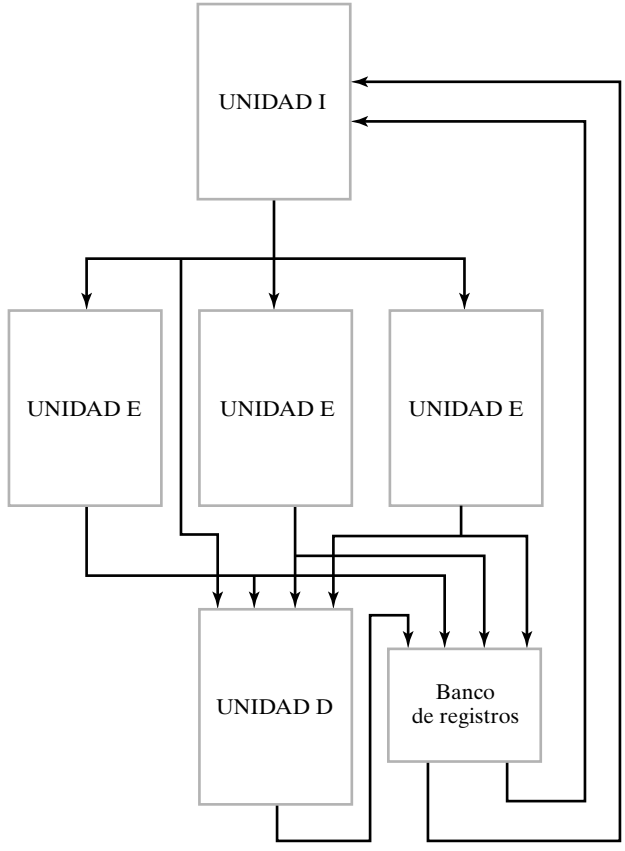
Los dos diseños que se han considerado en este capítulo representan dos diferentes ISA y dos organizaciones diferentes que soporta la CPU. La arquitectura RISC cuadra bien con la organización del control en *pipeline* por la simplicidad de las instrucciones. Debido a la necesidad de tener un rendimiento alto, la moderna arquitectura presentada se construye sobre la base de un RISC. En esta sección, manejaremos características adicionales para acelerar el RISC segmentado de base. Finalmente, relacionaremos las dos organizaciones con un diseño más general de los sistemas digitales.

Conceptos de CPU de alto rendimiento

Entre los diversos métodos utilizados para diseñar CPUs de alta velocidad están las unidades múltiples organizadas en una estructura paralela en *pipeline*, *superpipelines*, y las arquitecturas superescales.

Considere el caso en el que una operación necesita varios ciclos de reloj para ejecutarse, pero que las operaciones de acceso a la instrucción y la reescritura se pueden realizar en un solo ciclo de reloj. Entonces es posible iniciar una instrucción cada ciclo de reloj pero no es posible completar la ejecución de una instrucción en cada ciclo. En dicha situación, el rendimiento de la CPU puede mejorarse sustancialmente teniendo varias unidades de ejecución en paralelo. En la Figura 12-22 se muestra un diagrama de alto nivel de bloques para este tipo de sistemas. El acceso de la instrucción, decodificación y acceso del operando se puede hacer en la unidad I del *pipeline*. Además, la unidad I puede manejar bifurcaciones. Cuando la decodificación de una instrucción distinta a la de bifurcación se ha completado, la instrucción y los operandos se mandan a la unidad E adecuada. Cuando la ejecución de la instrucción se ha completado en la unidad E, se realiza la reescritura en el banco de registros. Si es necesario un acceso a memoria, entonces se utiliza la unidad D para ejecutar la escritura de la memoria. Si la operación es un almacenamiento, va directamente a la unidad D. Véase que las unidades de ejecución reales pueden ser microprogramadas y pueden tener también *pipelines* internos.

Suponga que tenemos una secuencia de tres instrucciones —por ejemplo, una multiplicación, un desplazamiento de 16 bits y una suma— sin conflictos de datos. Suponga además que hay una sola unidad E en *pipeline* que realiza todas estas operaciones, que necesitan 17, 8 y 2 ciclos de reloj, respectivamente, y que tanto la multiplicación como el desplazamiento necesitan

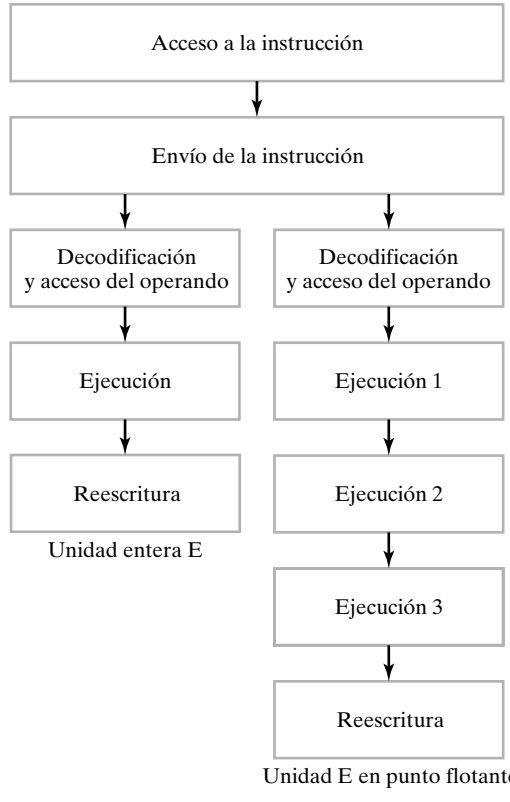


□ **FIGURA 12-22**
Organización en varias unidades de ejecución

varios pasos a través de las partes de la unidad E en *pipeline*. Esta situación sólo permite un solapamiento de un ciclo de reloj entre las parejas de las tres instrucciones. De esta forma, lo más rápido que se ejecuta la secuencia de operaciones en la unidad E es con $17 + 8 + 2 - 2 = 25$ ciclos de reloj. Pero con una unidad E para cada operación, éstas se pueden ejecutar en el máximo de (17, 1 + 8, 2 + 2) ciclos de reloj, que es igual a 17 ciclos de reloj. El 1 y el 2 adicionales se deben al envío de una instrucción por ciclo de reloj a la unidad E seleccionada. El *throughput* resultante de la ejecución se ha mejorado en un factor de $25/17 = 1.5$.

En todos los métodos considerados hasta ahora, el pico posible en el *throughput* es de una instrucción por ciclo de reloj. Con esta limitación, es deseable maximizar la velocidad de reloj minimizando el retardo máximo de la etapa segmentada. Si, como consecuencia, se utiliza un número grande de etapas en el pipeline, se dice que la CPU esta *supersegmentada* (en inglés *superpipelined*). Una CPU supersegmentada tendrá, generalmente, una frecuencia de reloj muy alta, en el rango de los GHz. Sin embargo, en dicha organización, el manejo efectivo de los conflictos es crítica, puesto que una parada o reinicialización del *pipeline* degrada el rendimiento de la CPU significativamente. Además, cuantas más etapas se añaden al *pipeline*, más dividida queda la lógica combinacional, y los tiempos de *setup* y de propagación de los flip-flops empiezan a dominar los retardos entre los dos registros que definen una etapa del *pipeline* y la velocidad de reloj. La mejora conseguida es menor y, cuando se tienen en cuenta los conflictos, el rendimiento real puede empeorarse en lugar de mejorarse.

Para ejecuciones rápidas, una alternativa a la supersegmentación es el uso de una organización *superescalar*. El objetivo de este tipo de organización es tener un pico de velocidad en la inicialización de las instrucciones que exceda a una instrucción por ciclo de reloj. En la Figura 12-23 se muestra una CPU superescalar que accede simultáneamente a parejas de instrucciones usando una ruta de datos para palabras dobles de la memoria. El procesador chequea los conflictos entre las instrucciones, así como la disponibilidad de las unidades de ejecución en la etapa de envío de la instrucción del *pipeline*. Si hay conflictos o las correspondientes unidades de ejecución ocupadas para la primera instrucción, entonces se retienen ambas instrucciones para enviarlas más tarde. Si la primera instrucción no tiene conflictos y su unidad E está disponible, pero hay un conflicto o no está disponible una unidad E para la segunda instrucción, entonces sólo se envía la primera instrucción. En caso contrario, se envían las dos instrucciones en paralelo. Si una determinada arquitectura superescalar tiene la capacidad de enviar hasta cuatro instrucciones simultáneamente, el pico en la tasa de ejecución es de cuatro instrucciones por ciclo de reloj. Si el ciclo de reloj es de 5 ns, la CPU tiene un pico en la tasa de ejecución de 800 MIPS. Nótese que el chequeo de conflictos para las instrucciones en las etapas de ejecución y aquellas que están en la etapa de envío pasa a ser muy compleja, tanto como se incrementa el número máximo de instrucciones a enviar simultáneamente. La complejidad del hardware resultante presenta la posibilidad de incrementar el tiempo del ciclo de reloj, de forma que hay que examinar muy cuidadosamente el intercambio de información en estos diseños.



□ **FIGURA 12-23**
Organización superescalar

Cerraremos esta sección con dos observaciones. Primero, como la búsqueda de un mejor rendimiento provoca organizaciones con un diseño cada vez más complejo, los conflictos hacen que el orden de las instrucciones juegue un papel más importante en el *throughput* que se puede conseguir. Además, se pueden conseguir mejoras en el rendimiento reduciendo el número de instrucciones que generan conflictos, como las bifurcaciones. Como consecuencia de esto, para explotar el rendimiento de las capacidades del hardware, el programador de lenguaje ensamblador y de compiladores necesitar tener un amplio conocimiento del comportamiento, no sólo de la arquitectura de conjunto de instrucciones, sino de la organización que subyace al hardware de la CPU.

Cuando hay involucradas varias unidades de ejecución, con mucha frecuencia, el diseño de CPUs que hemos estado considerando aquí, pasa a ser realmente el diseño del procesador completo, como se mostró para la computadora genérica. Esto es evidente en la organización superescalar de la Figura 12-23, que contiene una unidad de punto flotante (FPU). La FPU, la MMU y la parte de la caché interna que maneja los datos son, efectivamente, cuatro tipos de unidades E. La parte de la caché interna que maneja las instrucciones se puede ver como parte de la unidad I que accede a las instrucciones. De esta forma, en la búsqueda de mayor y mayor *throughput*, el dominio de la CPU pasa al del procesador, como en la computadora genérica.

Recientes innovaciones arquitecturales

Detrás de los conceptos presentados en las secciones anteriores, hay dos tendencias generales que parecen evidentes en una de las arquitecturas más recientes de alto rendimiento. La primera tendencia es el desarrollo de compiladores y de arquitecturas hardware que permitan al compilador identificar explícitamente las instrucciones hardware que se pueden ejecutar en paralelo. En este método, la identificación del paralelismo, hecha típicamente en el hardware de las arquitecturas superescalares, se ha trasladado ahora, en cierto grado, hacia el compilador. Esto descarga el hardware para otros usos, principalmente más unidades de ejecución y bancos de registros más grandes. La segunda tendencia es el uso de técnicas que permiten al procesador evitar que se hagan esperas en las bifurcaciones y que estén disponibles los valores de los datos. Estudiaremos tres técnicas que soportan esta tendencia en esta sección.

En lugar de esperar a que se realice una bifurcación, el procesador ejecuta las dos posibilidades de la bifurcación y genera ambos resultados. Cuando los resultados de la bifurcación están disponibles, se selecciona el resultado correcto y prosigue su cálculo. De esta forma, no hay retardo de espera en la bifurcación, mejorando significativamente el rendimiento de los *pipelines* largos. A este sencillo método se le denomina *predicción* y utiliza registros especiales de 1 bit, llamados registros de *predicción*, que determinan qué resultado se usa cuando se conoce el resultado de la bifurcación.

En lugar de esperar para cargar de datos de la memoria hasta que se conozca qué dato se necesita, se realiza una *carga especulativa* del dato de la memoria antes de que se conozca con seguridad el dato necesario. La razón del uso de esta técnica es evitar retardos relativamente largos necesarios para acceder a un operando de la memoria. Si el dato que se accede especulativamente es el dato que se necesita, entonces el dato estará disponible y el cálculo puede proseguir inmediatamente sin tener que esperar a un acceso a memoria para conseguir el dato.

En lugar de esperar a que el dato esté disponible, la *especulación de datos* utiliza métodos que predicen los valores de los datos y procede a realizar cálculos utilizando estos valores. Cuando se conoce el valor real y coincide con el valor predicho, entonces el resultado producido a partir del valor predicho se puede usar para seguir avanzando en los cálculos. Si el valor

real y el valor predicho difieren, entonces el resultado basado en el valor predicho se descarta y se usa el valor real para continuar los cálculos. Un ejemplo de especulación de datos es permitir que un valor se cargue de la memoria antes de que ocurra pronto un almacenamiento en la misma posición de memoria en el programa que se está ejecutando. En este caso, se predice que el almacenamiento no cambiará el valor del dato en la memoria, de forma que el valor cargado anteriormente será válido. Si, al mismo tiempo ocurre el almacenamiento, el valor cargado no es válido, el resultado del cálculo se descarta.

Todas estas técnicas realizan operaciones o secuencias de operaciones en las que los resultados se descartan con cierta frecuencia. De esta forma, hay un «gasto» de cálculo. Para poder hacer grandes cantidades de cálculos útiles, así como gasto de cálculo, se necesitan más recursos en paralelo, así como hardware especializado para llevar a cabo estas técnicas. El beneficio que se obtiene a cambio del coste de estos recursos es un rendimiento potencialmente más alto.

Sistemas digitales

Los dos diseños de sistemas digitales que hemos examinado en este capítulo son CPUs de propósito general ¿Cómo se puede relacionar su diseño con el de otros sistemas digitales? Antes de nada, cada sistema digital tiene su arquitectura. Aunque esa arquitectura no tenga que manejar instrucciones en ningún caso, es probable que, aún así, se pueda describir mediante transferencia de registros y, posiblemente, con una o más máquinas de estados. Por otra parte, puede tener instrucciones, pero pueden ser muy diferentes de las de una CPU. El sistema puede que no tenga ruta de datos o puede que tenga varias rutas de datos. Probablemente tendrá alguna forma de unidad de control y podrá tener varias unidades de control que interactúen. El sistema puede incluir o no memorias. De esta forma, el espectro total de los sistemas digitales tiene un amplio rango de posibilidades arquitecturales.

Entonces ¿cuál es la conexión de un sistema digital general con el contenido de este capítulo? Simplemente expone que la conexión son técnicas de diseño. Como ilustración, consideremos que hemos mostrado en detalle cómo se puede hacer un sistema con instrucciones utilizando una ruta de datos y una unidad de control. A partir de aquí, es relativamente fácil realizar un sistema más simple sin instrucciones. Hemos mostrado qué velocidades se pueden conseguir utilizando *pipelines* o unidades de ejecución en paralelo. De esta forma, si la meta de un sistema es alta velocidad, la segmentación, o *pipelining*, o unidades trabajando en paralelo son técnicas que hay que tener en consideración. Por ejemplo, uno de los autores, en un ejemplo de diseño de un sistema para hacer una parte de un transmisor USB (*véase* Sección 13-4), utiliza una ruta de datos en *pipeline* con un control que involucra tanto control en *pipeline* como un control convencional. Hemos mostrado cómo se puede utilizar la microprogramación para realizar controles de funciones complejas llevadas a cabo en un *pipeline*. Si un sistema tiene una o más funciones complejas, ya sea en *pipeline*, programable o no, el control microprogramado es una posibilidad a considerar.

12-6 RESUMEN

Este capítulo ha cubierto el diseño de dos procesadores —uno para un procesador de conjunto reducido de instrucciones (RISC) y procesador de conjunto de instrucciones complejo (CISC). Como preludio del diseño de estos procesadores, el capítulo comenzó con la ilustración de una ruta de datos segmentada o en *pipeline*. El concepto de *pipeline* posibilita realizar operaciones a frecuencias de reloj y con un *throughput* que no son alcanzables con los mismos componentes

en una ruta de datos convencional. Se presentó el diagrama de patrones de ejecución del *pipeline* para visualizar el comportamiento del *pipeline* y estimar así su pico de rendimiento. El problema de la baja frecuencia de reloj de un procesador de un solo ciclo se enfocó añadiendo una unidad de control en *pipeline* a la ruta de datos.

A continuación examinamos el diseño de un RISC con la ruta de datos y la unidad de control segmentada. Basándonos en el procesador de un solo ciclo de reloj del Capítulo 10, se caracterizó la ISA RISC mediante instrucciones de longitud sencilla, un número limitado de instrucciones con sólo algunos modos de direccionamiento con accesos de memoria restringidos a operaciones de carga y almacenamiento. La mayoría de las operaciones RISC son sencillas en el sentido de que, en una arquitectura convencional, se pueden ejecutar utilizando una sola microoperación.

La ISA RISC se realiza mediante el uso de una versión modificada de la ruta de datos segmentada de la Figura 12-2. Las modificaciones incluyen un incremento en la longitud de palabra a 32 bits, doblando el número de registros en el banco de registros y reemplazando el desplazador de la unidad funcional por un *barrel shifter*. Asimismo, se utiliza la versión modificada de la unidad de control de la Figura 12-4. Los cambios en el control se hicieron para acomodar los cambios de la ruta de datos y para manejar las bifurcaciones y saltos en un entorno con *pipeline*. Después de terminar con las bases del diseño, se dieron consideraciones para los problemas de conflictos de datos y de control. Examinamos cada tipo de conflicto, así como las soluciones software y hardware para cada una.

La ISA de un CISC tiene el potencial de realizar muchas operaciones distintas, con accesos a memoria que soportan varios modos de direccionamiento. El CISC también tiene operaciones que son complejas en el sentido de que requieran muchos ciclos de reloj para su ejecución. El CISC permite que muchas instrucciones puedan realizar accesos a la memoria y se caracterizan por bifurcaciones condicionales complejas soportadas por códigos de condición (bits de status). Aunque, en general, una ISA CISC permite instrucciones de longitud múltiple, esta característica no se proporciona en la arquitectura de ejemplo.

Para proporcionar altos *throughput*, la arquitectura RISC sirve como corazón para la arquitectura CISC. Las instrucciones sencillas se pueden ejecutar con el *throughput* de un RISC, con instrucciones complejas, ejecutadas en varios pasos a través del RISC en *pipeline*, reduciendo el *throughput* total. La modificación de la ruta de datos del RISC proporcionó un registro para almacenar operandos temporalmente y códigos de condición. Los cambios en la unidad de control fueron necesarios para soportar los cambios en la ruta de datos. La principal modificación de la unidad de control fue, sin embargo, añadir el control microprogramado para la ejecución de instrucciones complejas. Los cambios añadidos a la unidad de control del RISC fueron necesarias para integrar el control del microprograma en el control del *pipeline*. Se dieron ejemplos de microprogramas para tres instrucciones complejas.

Después de terminar el diseño del CISC y del RISC, tocamos algunos conceptos avanzados, incluyendo unidades de proceso en paralelo, CPUs supersegmentadas, CPUs superescalares y técnicas predictivas y especulativas para conseguir altos rendimientos. Para terminar, hemos relacionado las técnicas de diseño de este capítulo para el diseño de sistemas digitales en general.

REFERENCIAS

1. MANO, M. M.: *Computer System Architecture*, 3rd Ed. Englewood Cliffs, NY: Prentice Hall, 1993.

2. PATTERSON, D. A., and J. L. HENNESSY: *Computer Organization and Design: The Hardware/Software Interface*, 2nd ed. San Francisco, CA: Morgan Kaufmann, 1998.

3. HENNESSY, J. L., and D. A. PATTERSON: *Computer Architecture: A Quantitative Approach*, 2nd ed. San Francisco, CA: Morgan Kaufmann, 1996.

4. DIETMEYER, D. L.: *Logic Design of Digital Systems*, 3rd ed. Boston, MA: Allyn-Bacon, 1988.

5. KANE, G., And J. HEINRICH: *MIPS RISC Architecture*. *Englewood Cliffs, NJ*: *Prentice Hall*, 1992.

6. SPARC INTERNATIONAL, INC.: *The SPARC Architecture Manual: Version 8.* Englewood Cliffs, NJ: Prentice Hall, 1992.

7. WEISS, S., And J. E. SMITH: *POWER and PowerPC*. San Mateo, CA: Morgan Kaufmann, 1994.

8. WYANT, G., and T. HAMMERSTROM: *How Microprocessors Work*. Emeryville, CA: Ziff-Davis Press, 1994.

9. HEURING, V., and H. JORDAN: *Computer Systems Design and Architecture*. Upper Saddle River, NJ: Prentice-Hall,1997.

PROBLEMAS

El signo (+) indica problemas más avanzados y el asterisco indica que hay una solución disponible en la dirección de Internet: http://www.librosite.net/Mano.

12-1. Una ruta de datos en *pipeline* es similar a la de la Figura 12-1(b) pero con los retardos, desde arriba hasta abajo, sustituidos por los siguientes valores: 1.0 ns, 1.0 ns, 1.0 ns, 0.2 ns y 0.1 ns. Determine (a) la frecuencia máxima de reloj, (b) el tiempo de latencia y (c) el *throughput* máximos para esta ruta de datos.

12-2. *Un programa consistente en una secuencia de 12 instrucciones sin instrucciones de bifurcación ni de salto se ejecuta en un procesador con un *pipeline* de seis etapas con un periodo de reloj de 1.25 ns. Determine (a) el tiempo de latencia del *pipeline*, (b) el *throughput* máximo del *pipeline* y (c) el tiempo necesario para la ejecución del programa.

12-3. Se accede y se ejecuta la secuencia de siete instrucciones LDI en el programa de número de registro con el patrón de ejecución del *pipeline* dado en la Figura 12-5. Simule manualmente la ejecución dando, por cada ciclo de reloj, los valores en los registros *PC*, *IR*, Data *A*, Data *B*, Data *F*, Data *I* y en el banco de registros, teniendo sus valores cambiados para cada ciclo de reloj. Suponga que todo el banco de registro contiene -1 (todo 1s).

12-4. Para cada una de las operaciones RISC de la Tabla 12-1, enumere el modo o modos de direccionamiento utilizados.

12-5. Simule la operación del *barrel shifter* de la Figura 12-8 para cada uno de los siguientes desplazamientos y $A = 7E93C2A1_{16}$. Enumere los valores en hexadecimal sobre las 47 líneas, 35 líneas y 32 líneas de salida de los tres niveles del desplazador.

(a) Izquierda, $SH = 11$ **(b)** Derecha, $SH = 13$ **(c)** Izquierda, $SH = 30$

12-6. *En la CPU RISC de la Figura 12-9, simule manualmente, en hexadecimal, el proceso de la instrucción ADI R1 R16 2F01 localizada en $PC = 10F$. Suponga que $R16$ contiene

el valor 0000001F. Muestre el contenido de cada registro del *pipeline* y del banco de registros (en este último sólo cuando cambie) para cada ciclo de reloj.

12-7. Repita el Problema 12-6 para las instrucciones SLT R31 R10 R16 con R10 conteniendo 0000100F y R16 conteniendo 00001022.

12-8. Repita el Problema 12-6 para la instrucción LSL R1 R16 000F.

12-9. +Utilice un programa de computadora de minimización de lógica para diseñar el decodificador de instrucciones de un RISC a partir de la Tabla 21-3. No se necesita hacer el campo FS ya que se puede cablear directamente del OPCODE.

12-10. *Para el diseño RISC, dibuje el diagrama de ejecución del siguiente programa RISC e indique cualquier conflicto de datos que se presente:

```
1 MOVA   R7, R6
2 SUB    R8, R8, R6
3 AND    R8, R8, R7
```

12-11. Para el diseño RISC, dibuje el diagrama de ejecución del siguiente programa RISC (con los contenidos de *R*7 distinto de cero después de la resta), e indique cualquier conflicto de datos o de control que se presente:

```
1 SUB    R7, R7, R6
2 BNZ    R7, 000F
3 AND    R8, R7, R6
4 OR     R5, R8, R5
```

12-12. *Rescriba el programa RISC del Problema 12-10 y el Problema 12-11 utilizando NOPs para evitar todos los conflictos de datos y de control y dibuje los nuevos diagramas de ejecución.

12-13. Dibuje los diagramas de ejecución del programa del Problema 12-10, suponiendo que:

(a) la CPU RISC con parada de datos dada en la Figura 12-12.
(b) la CPU RISC con anticipación de datos de la Figura 12-13.

12-14. Simule el procesamiento del programa del Problema 12-11 utilizando la CPU RISC con parada por conflicto de datos de la Figura 12-12. Dé el contenido de cada registro del *pipeline* y del banco de registros (de este último sólo cuando ocurra un cambio) para cada ciclo de reloj. Inicialmente, *R*6 contiene 00000010_{16}, *R*7 contiene 00000020_{16}, *R*8 contiene 00000030_{16} y el *PC* contiene 00000001_{16}. ¿Se evita el conflicto de datos?

12-15. *Repita el Problema 12-14 utilizando la CPU RISC con anticipación de datos de la Figura 12-13.

12-16. Dibuje el diagrama de ejecución del programa del Problema 12-11, suponiendo la combinación de una CPU RISC con predicción de bifurcación de la Figura 12-17 y la CPU RISC con adelantamiento de datos de la Figura 12-13.

12-17. Diseñe la Unidad de Constantes en la CPU CISC en *pipeline* utilizando la información dada en la Tabla 12-5 y multiplexores de buses, puertas AND, OR e inversores.

12-18. *Diseñe la lógica del registro de direcciones de la CPU CISC utilizando la información dada en los campos de registros de la Tabla 12-5 más multiplexores de buses, puertas AND, OR e inversores.

12-19. Diseñe la Lógica de Control de Direcciones descrita en la Tabla 12-4 utilizando puertas AND, OR e inversores.

12-20. Escriba el microcódigo para la parte de ejecución de las siguientes instrucciones CISC. Dé la descripción de las transferencias de registros y las representaciones en binario o hexadecimal similar a la mostrada en la Tabla 12-6 con el código binario de cada instrucción.

(a) Comparación mayor que
(b) Bifurcación si es menor que cero (bit N = 1 del *CC*)
(c) Bifurcación si hay *overflow* (bit V = 1 del *CC*)

12-21. Repita el Problema 12-20 para las siguientes instrucciones CISC que se especifican mediante sentencia de transferencias de registro.

(a) Push: $R[SA] \leftarrow R[SA] + 1$ seguida por: $M[R[SA]] \leftarrow R[SB]$
(b) Pop: $R[DR] \leftarrow M[R[SA]]$ seguida por: $R[SA] \leftarrow R[SA] - 1$

12-22. *Repita el problema 12-21 para las siguientes instrucciones CISC.

(a) Suma con acarreo: $R[DR] \leftarrow R[SA] + R[SB] + C$
(b) Resta con acarreo: $R[DR] \leftarrow R[SA] - R[SB] - B$

12-23. Repita el Problema 12-21 para las siguientes instrucciones CISC.

(a) Suma con memoria indirecta: $R[DR] \leftarrow R[SA] + M[M[R[SB]]]$
(b) Suma a la memoria: $M[R[DR]] \leftarrow M[R[SA]] + R[SB]$

12-24. *Repita el Problema 12-20 para la instrucción CISC, suma escalar de memoria. Esta instrucción utiliza el contenido de *R*[SB] como longitud del vector. Se suman los elementos del vector con su elemento menos significativo de la memoria apuntada por *R*[SA] y coloca el resultado en la posición de memoria apuntada por *R*[DR].

12-25. Repita el Problema 12-20 para la instrucción CISC, suma vectorial de memoria. Esta instrucción utiliza el contenido de *R*[SB] como longitud del vector. Se suma el vector con su elemento menos significativo en la memoria apuntado por *R*[SA] al vector con su elemento menos significativo en la memoria apuntado por *R*[DR]. El resultado de la adición sustituye al vector con su elemento menos significativo apuntado por *R*[DR].

CAPÍTULO

13

ENTRADA/SALIDA Y COMUNICACIONES

En este capítulo damos una visión global de algunos aspectos de los procesadores de entrada y salida (E/S) y comunicaciones entre la CPU y los dispositivos de E/S, interfaces E/S y procesadores de E/S. Debido a la amplia gama de dispositivos de E/S y la demanda en la manipulación rápida de programas y datos, la E/S es una de las áreas más complejas del diseño de procesadores. Como consecuencia de esto, estamos en disposición de presentar algunas partes seleccionadas de este puzzle. Sólo ilustraremos con detalle tres dispositivos: un teclado, un disco duro y una tarjeta gráfica. Posteriormente presentaremos el bus de E/S y las interfaces de E/S que conectan los dispositivos de E/S. Consideraremos como ejemplo las comunicaciones serie y el uso de la estructura de un teclado. Luego veremos el bus universal serie (*Universal Serial Bus*, USB) y una solución alternativa para el problema del acceso a los dispositivos E/S. Finalmente, estudiaremos los cuatro modos de realizar transferencias de datos: transferencia controlada por programa, transferencia iniciada por interrupción, acceso directo a memoria y el uso de procesadores de E/S.

En términos de la computadora genérica al comienzo del Capítulo 1, es evidente que la E/S involucra a una gran parte de la computadora. No están muy involucrados el procesador, la caché externa y la RAM, aunque éstos también se usan extensivamente para dirigir y realizar las transferencias de E/S. Incluso la computadora genérica, que tiene menos dispositivos de E/S que la mayoría de los PC, tiene varios de estos dispositivos que necesitan de un hardware digital importante que los realice.

13-1 PROCESADORES DE E/S

El subsistema de entrada y salida de un procesador proporciona un modo eficiente de comunicación entre la CPU y el entorno exterior. Los programas y los datos deben introducirse en la memoria para procesarse y los resultados obtenidos de los cálculos deben guardarse o ser mostrados. Entre los dispositivos de entrada y salida que se encuentran comúnmente en una computadora están teclados, monitores, impresoras, discos magnéticos y lectores de discos compactos de solo lectura (CD-ROM). Otros dispositivos de entrada y salida que se encuentran frecuentemente son los módems u otras interfaces para las comunicaciones, scanners y tarjetas de sonido con altavoces y micrófonos. Un número importante de procesadores, como los usados en automóviles, tienen conversores analógico-digitales, conversores digitales-analógicos y otros sistemas de adquisición de datos y componentes de control.

La interfaz de E/S de un procesador está en función de una determinada aplicación. Esto da lugar a una amplia gama de dispositivos y sus correspondientes diferencias en las necesidades para interactuar con ellos. Puesto que cada dispositivo se comporta de forma diferente, se podría consumir mucho tiempo si nos paramos a ver con detalle las interconexiones necesarias entre el procesador y cada periférico. Examinaremos, por tanto, sólo tres periféricos que aparecen en la mayoría de las computadoras: el teclado, el disco duro y la tarjeta gráfica. Estos representan los puntos típicos en el rango de la tasa de transferencia de datos necesaria para los periféricos. Además, presentamos algunas de las características comunes encontradas en los subsistemas de E/S de los procesadores, así como las diferentes técnicas para las transferencias de datos tanto en paralelo, usando varias rutas de trasporte, como en serie, a través de líneas de comunicación.

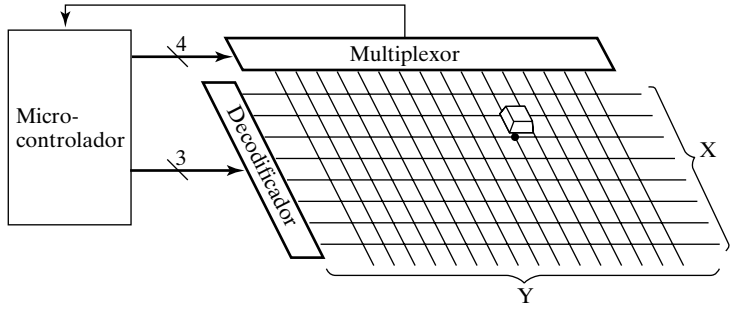
13-2 EJEMPLO DE PERIFÉRICOS

Los dispositivos controlados directamente por la CPU se dice que están conectados en línea. Estos dispositivos se comunican directamente con la CPU o transfieren la información binaria dentro y fuera de la memoria bajo la dirección de la CPU. A los dispositivos de entrada y salida conectados en línea al procesador se les llama *periféricos*. En esta sección, estudiaremos tres dispositivos periféricos: un teclado, un disco duro y una tarjeta gráfica. Utilizaremos el teclado para ilustrar los conceptos de E/S en una sección posterior. Presentaremos el disco duro para ver la necesidad del acceso directo a memoria y proporcionar una base para ver el papel que juega el dispositivo en el Capítulo 14 como componente en una memoria jerárquica. Incluimos la tarjeta gráfica para ilustrar el altísimo potencial de los requerimientos de alta tasa de transferencia de datos en las aplicaciones actuales.

Teclado

El teclado es uno de los dispositivos electromecánicos más sencillos que se conectan típicamente a una computadora. Puesto que se maneja manualmente, tiene una de las tasas de transferencia de datos más baja de entre los periféricos.

El teclado está compuesto por un conjunto de teclas que se presionan por el usuario. Es necesario detectar qué tecla se ha pulsado. Para hacerlo se utiliza una *matriz de rastreo* (en inglés *scan matrix*) que yace debajo de las teclas, como se muestra en la Figura 13-1. Esta matriz bidimensional es conceptualmente similar a la matriz utilizada en la memoria RAM. La matriz mostrada en la figura es de 8 × 16, dando lugar a 128 intersecciones, de forma que puede manejar



□ **FIGURA 13-1**
Matriz de rastreo de un teclado

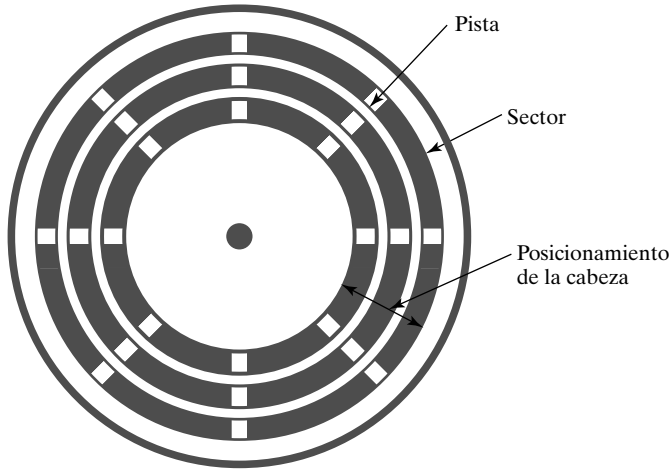
hasta 128 teclas. Un decodificador gobierna las líneas *X* de la matriz, que son análogas a las líneas de palabra de una memoria RAM. Se conecta un multiplexor a las líneas Y de la matriz, que son análogas a las líneas de bit de una memoria RAM. El decodificador y el multiplexor se controlan mediante un microcontrolador, un pequeño procesador que contiene memoria RAM, ROM, un temporizador y sencillas interfaces de E/S.

El microcontrolador se programa para rastrear periódicamente todas las intersecciones de la matriz manipulando las entradas de control del decodificador y el multiplexor. Si se pulsa una tecla de una intersección, se cierra el camino de una señal desde una salida del decodificador *X* hasta una entrada del multiplexor *Y*. La existencia de este camino se percibe en una de las entradas del microcontrolador. El código de control, de 7 bit, aplicado al decodificador y al multiplexor al mismo tiempo, identifica la tecla. Para permitir el «refinamiento» en el tecleo, en el que varias teclas se pulsan antes de que otras se dejen de pulsar, en realidad, el microcontrolador identifica el pulsado y la liberación (dejar de pulsar) de las teclas. Si una tecla se pulsa o se libera, el código de control en el instante del evento se percibe y se traduce por el microcontrolador a un código que llamaremos *código scan-K*. Cuando se pulsa una tecla, se genera un *código de marca*; cuando se deja de pulsar, se produce un *código de ruptura*. De esta forma se crean dos códigos para cada tecla, uno para cuando se presiona la tecla y otro para cuando se suelta. Nótese que el muestreo de todo el teclado ocurre centenares de veces en un segundo, de forma que no hay peligro de perder ninguna pulsación ni ninguna liberación de una tecla.

Después de presentar varios conceptos de las interfaces de E/S, volveremos a visitar el teclado para ver qué sucede con los códigos de *scan-K* antes de que se traduzca definitivamente a caracteres ASCII.

Disco duro

El disco duro es el principal medio de almacenamiento de escritura, no volátil, de velocidad intermedia en la mayoría de las computadoras. El disco duro almacena la información en serie sobre un disco fijo con algunos o muchos platos, como se ve en la esquina superior derecha de la computadora genérica del Capítulo 1. Cada plato es magnetizable en una o en ambas superficies. Hay una o más *cabezas* de lectura/escritura por cada superficie gravable; para el resto de nuestro estudio supondremos que hay una sola cabeza por superficie. Cada plato se divide en *pistas* concéntricas, como se ilustra en la Figura 13-2. Al conjunto de pistas que están a la misma distancia del centro de disco en todos los platos se les llama *cilindro*. Cada pista se divide en *sectores* que contienen un número fijo de bytes. El número de bytes por sector varía entre 256



□ **FIGURA 13-2**
Formato del disco duro

y 5 K. El byte de dirección típico contiene el número del cilindro, el número de cabeza, el número de sector y el desplazamiento de la palabra dentro del sector. El direccionamiento supone que el número de sectores por pista es fijo. Actualmente, los discos de alta capacidad tienen más sectores en las pistas externas, que son más largas, y menos sectores en las pistas interiores, que son más cortas. Además, se reservan un cierto número de sectores que tomarán el lugar de sectores defectuosos. Como consecuencia de estas elecciones de diseño, la dirección física del sector en uso del disco es, probablemente, diferente de la dirección mandada por el controlador del disco. La correspondencia de esta dirección con la dirección física se lleva a cabo por el controlador del disco u otro *driver* electrónico.

Para permitir acceder a la información, el conjunto de cabezas se montan sobre un servomotor que mueve las cabezas radialmente sobre el disco, como se muestra en la computadora genérica dibujado. Al tiempo necesario para mover las cabezas desde un cilindro hasta el deseado se llama *tiempo de búsqueda* (*seek time*). Al tiempo necesario para girar el disco desde su posición actual hasta la que tiene el sector deseado bajo las cabezas se llama *retardo de giro* (*rotational delay*). Además, el controlador del disco necesita una cierta cantidad de tiempo para acceder y sacar la información. Este tiempo es el *tiempo del controlador* (*controller time*). Al tiempo necesario para localizar una palabra sobre el disco es el *tiempo de acceso* (*access time*), que es la suma del tiempo del controlador, el tiempo de búsqueda y el retador de giro. Para estos cuatros parámetros se utilizan valores medios. Las palabras se pueden transferir una a una pero, como se verá en el capítulo 14, con frecuencia se acceden a ellas en bloques. La tasa de transferencia para los bloques de palabras, una vez que el bloque ha sido localizado, es la *tasa de transferencia del disco*, especificada típicamente en megabytes/segundo (MB/s). La tasa de transferencia requerida por el bus CPU-memoria para transferir un sector del disco es el número de bytes en el sector dividido por la cantidad de tiempo tomado para leer un sector del disco. La cantidad de tiempo necesario para leer un sector es igual a la proporción del cilindro ocupado por el sector dividido por la velocidad de rotación del disco. Por ejemplo, con 63 sectores, 512 Bytes por sector, una velocidad de rotación de 5400 rpm, y permitiendo una separación entre sectores, este tiempo es de 0.15 ms aproximadamente, dando una tasa de transferencia de 512/0.15 ms = 3.4 MB/s. El controlador almacenará la información leída del sector en su memoria. La suma del tiempo de acceso al disco y la tasa de transferencia de disco multiplicada por el número de

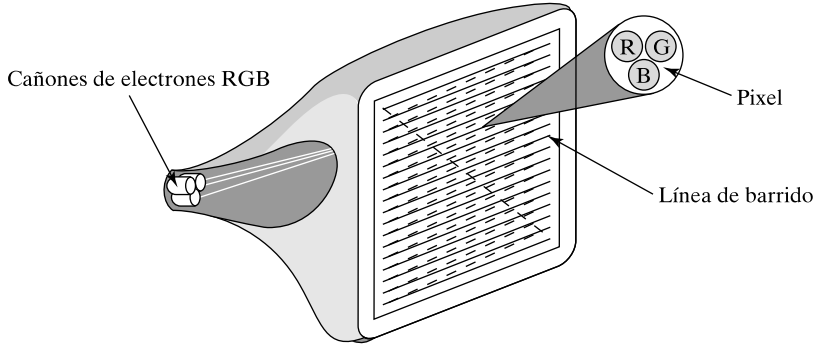
bytes por sector da una estimación del tiempo necesario para transferir la información en un sector a o desde el disco duro. Los valores típicos a mediados de los años 90 eran un tiempo de búsqueda de 10 ms, una velocidad de rotación de 6 ms, un tiempo de transferencia de un sector de 0.15 ms y un tiempo del controlador despreciable, dando un tiempo de acceso para un sector aislado de 16.15 ms.

Monitores gráficos

Los monitores gráficos o displays son los principales dispositivos de salida para el uso interactivo de una computadora. Los displays utilizan diversas tecnologías, la más corriente es la de tubo de rayos catódicos (CRT, del inglés *cathode-ray tube*), como se ilustra en la Figura 13-3. La mayoría de las versiones modernas de los displays CRT se basan en señales analógicas que se generan en la tarjeta adaptadora gráfica. El monitor se define en términos de dibujos elementales llamados *pixels*. El monitor de color tiene tres puntos asociados con cada píxel de la pantalla. A estos puntos les corresponden los colores primarios rojo, verde y azul (RGB, del inglés *red*, *green* y *blue*). En cada punto hay un fósforo coloreado. Un fósforo emite luz de su color cuando se excita con un haz de electrones. Para excitar a los tres fósforos simultáneamente se utilizan tres cañones de electrones, uno para el rojo, uno para el verde y otro para el azul, de aquí los cañones de electrones RGB mostrados en la figura. El color que resulta en un determinado píxel se determina por la intensidad del haz de electrones que golpean los fósforos dentro de cada píxel.

Los haces de electrones barren la pantalla formando un conjunto de líneas llamadas *líneas de barrido*. A este conjunto de líneas se les denomina *trama*. Las líneas se barren desde arriba hasta abajo, empezando por la esquina superior izquierda y terminando en la esquina inferior derecha. Los cañones de electrones permanecen con intensidad cero cuando van de la derecha a la izquierda para preparar el siguiente barrido de línea. La resolución de la información mostrada se da en términos de número de *pixels* por línea de barrido y por el número de líneas de barrido en la trama. Los monitores de alta resolución (*super video graphics array*, SVGA) pueden tener unos 1280 *pixels* por línea y 1024 líneas por trama. Los haces de electrones barren toda la trama en 1/60 de segundo.

Cada píxel se controla mediante el adaptador gráfico. Un adaptador típico utiliza un byte para definir el color de un píxel. Como el byte tiene 8 bits, se pueden definir 256 colores en un



□ **FIGURA 13-3**
Monitor CRT

determinado instante. El byte no se lleva directamente al monitor si no que se selecciona 1 de los 256 registros del adaptador gráfico para definir el color. Cada registro tiene 20 bits o más, así que los 256 colores se pueden seleccionar de entre 1 millón de colores definiendo el contenido de los registros.

Típicamente, los adaptadores gráficos tienen una memoria RAM de video que almacena todos los bytes que controlan los *píxels* del monitor. Para un monitor de alta resolución con 1280 *pixels* por línea y 1024 líneas de barrido, el número de píxeles es de $1280 \times 1024 = 1\,310\,720$. De esta forma, una sola pantalla de información necesita al menos 1.25 MB de memoria RAM de vídeo.

Tasas de transferencia de E/S

Como se indicó anteriormente, los tres dispositivos principales estudiados en esta sección dan una idea del rango de las tasas de transferencias de datos. La tasa de transferencia de datos de un teclado es menor que 10 bytes/s. En un disco duro, cuando el controlador de disco está capturando datos que llegan rápidamente del disco al buffer del sector, la transferencia de datos desde el buffer a la memoria principal es imposible. Así, en este caso en el que el siguiente sector se va a leer inmediatamente, todos los datos desde el *buffer* del sector necesitan ser almacenados en la memoria principal durante el tiempo en el que la separación entre sectores del disco pasa debajo de la cabeza del disco. Para 63 sectores y una velocidad de rotación de 5400 rpm, este tiempo es de 25 ms aproximadamente. De esta forma, el pico de la tasa de transferencia necesaria es de 512B/25 ms = 20 MB/s. En un monitor de 256 colores, si la pantalla se tiene que cambiar totalmente cada 1/60 de segundo, se ha de mandar 1.25 MB de datos a la memoria RAM de vídeo desde la CPU en esa cantidad de tiempo. La tasa de transferencia necesaria es $1.25 \text{ MB} \times 60 = 75$ MB/s.

Basándonos en lo anterior, podemos concluir que la tasa máxima de transferencia de datos necesaria para un periférico en concreto varía dentro de un amplio rango. Las tasas para el disco duro y el monitor son bastante altas comparadas con la máxima tasa de transferencia en los buses como para proporcionar un desafío a los diseñadores. Los esfuerzos para solventar este desafío utilizan técnicas en el controlador de disco duro y el adaptador gráfico para reducir las tasas máximas de transferencia y utilizar diseños de buses rápidos entre los interfaces de los periféricos y la memoria.

13-3 INTERFACES DE E/S

Los periféricos conectados a un procesador necesitan enlaces especiales de comunicación como interfaz entre ellos y la CPU. El propósito de estos enlaces es resolver las diferencias en las características de la CPU y la memoria y las características de cada periférico. La principales diferencias son:

1. Los periféricos son con frecuencia dispositivos electromecánicos cuya forma de operar es diferente a la de la CPU y la memoria, los cuales son dispositivos electromecánicos.
2. La tasa de transferencia de datos de los periféricos es, normalmente, diferente de la velocidad del reloj de la CPU. Como consecuencia, puede ser necesario un mecanismo de sincronización.
3. Los códigos de los datos y los formatos de los periféricos difieren del formato de las palabras de la CPU y de la memoria.

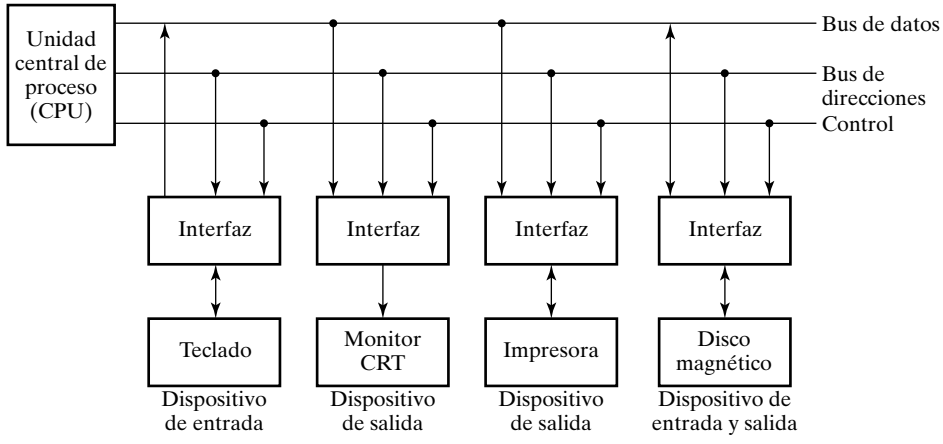
4. Los modos de operación de los periféricos difieren unos de otros y cada uno se debe controlar de forma que no perturbe la operación de los otros periféricos conectados a la CPU.

Para resolver estas diferencias, los sistemas basados en procesador incluyen componentes hardware especiales entre la CPU y los periféricos para supervisar y sincronizar todas las transferencias de entrada y salida. A estos componentes se les llama *unidades de interfaz* ya que conectan el bus de la CPU y el dispositivo periférico. Además, cada dispositivo tiene su propio controlador para supervisar las operaciones del mecanismo en concreto de cada periférico. Por ejemplo, el controlador de una impresora conectada a una computadora controla el movimiento del papel, el tiempo de la impresión y la selección de los caracteres a imprimir.

Unidad interfaz y bus de E/S

Una estructura típica de comunicación entre la CPU y los diversos dispositivos se muestra en la Figura 13-4. Cada periférico tiene una unidad de interfaz asociada a él. El bus común de la CPU se conecta a todas las interfaces de los periféricos. Para comunicarse con un dispositivo en particular, la CPU coloca la dirección de un dispositivo en el bus de direcciones. Cada interfaz conectada al bus contiene un decodificador de direcciones que monitoriza las líneas de dirección. Cuando una interfaz detecta su propia dirección se activa el camino entre las líneas del bus y el dispositivo que lo controla. Todos los periféricos con direcciones que no responden a la dirección del bus están deshabilitados por su interfaz. Al mismo tiempo que la dirección está disponible en el bus de direcciones, la CPU proporciona un código de función en las líneas de control. El interfaz seleccionado responde al código de función y procede a ejecutarlo. Si se deben transferir datos, la interfaz se comunica tanto con el dispositivo como con el bus de datos de la CPU para sincronizar la transferencia.

Además de comunicarse con los dispositivos de E/S, la CPU de un procesador debe comunicarse con la unidad de memoria a través de un bus de direcciones y otro de datos. Hay tres formas en que los buses externos de un procesador comunican la memoria y la E/S. Un método utiliza buses comunes de datos, direcciones y de control tanto para la memoria como para la E/S. Nos hemos referido a esta configuración como *E/S ubicada en memoria* (en inglés *memory-mapped I/O*). El espacio común de direcciones se comparte entre las unidades de interfaz y las



□ **FIGURA 13-4**
Conexión de los dispositivos de E/S a la CPU

palabras de la memoria, cada una teniendo diferentes direcciones. Los procesadores que adoptan este sistema de ubicación en memoria leen y escriben desde las unidades de interfaz como si hubiesen sido asignadas a las direcciones de memoria usando las mismas instrucciones que leen y escriben en la memoria.

La segunda alternativa es compartir un mismo bus de direcciones y un bus de datos pero utilizar diferentes líneas de control para la memoria y la E/S. Dichos procesadores tienen líneas separadas de lectura y escritura para la memoria y la E/S. Para leer o escribir en la memoria, la CPU activa el control de lectura o escritura de la memoria. Para realizar una entrada o una salida de una interfaz, la CPU activa el control de escritura y lectura de E/S usando instrucciones especiales. De esta forma, las direcciones asignadas a la memoria y a la interfaz de entrada y salida son independientes unas de otras y se distinguen mediante las líneas de control separadas. A este método se le denomina como *configuración aislada de E/S*.

La tercera alternativa es tener dos conjuntos de buses independientes para datos, direcciones y de control. Esto es posible en procesadores que incluyen un procesador de E/S en el sistema, además de la CPU. La memoria se comunica tanto con la CPU y el procesador de E/S a través de un bus de memoria común. El procesador de E/S se comunica con los dispositivos de E/S a través de líneas separadas de control, direcciones y datos. El propósito del procesador de E/S es proporcionar una ruta independiente para la transferencia de información entre los dispositivos externos y la memoria interna. Al procesador de E/S se le suele llamar *canal de datos*.

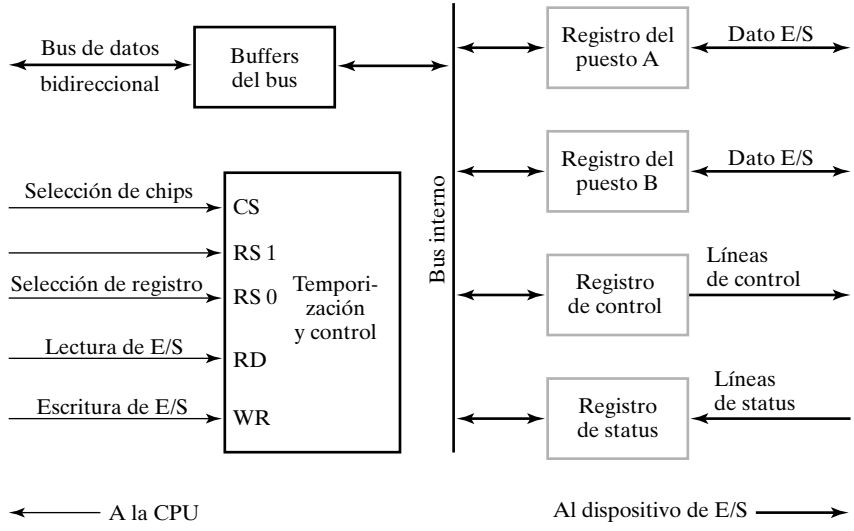
Ejemplo de interfaz de E/S

Una unidad de interfaz de E/S típica se muestra en el diagrama de bloques de la Figura 13-5. Consiste en dos registros de datos llamados *puertos*, un registro de control, un registro de status, un bus bidireccional y circuitos de control y de temporización. La función de la interfaz es trasladar las señales entre los buses de la CPU y los dispositivos de entrada y salida y proporcionar el hardware necesario para satisfacer las restricciones de tiempo de cada uno.

Los datos de E/S procedentes de los dispositivos se pueden transferir al puerto A o B. La interfaz puede funcionar con un dispositivo de salida, con un dispositivo de entrada o con un dispositivo que necesita tanto entrada como salida. Si la interfaz se conecta a una impresora, tendrá datos de salida, si sirve a un scanner tendrá datos de entrada. Un disco duro transfiere datos en ambas direcciones pero no al mismo tiempo, de manera que la interfaz sólo necesita un conjunto de líneas bidireccionales de datos de E/S.

El *registro de control* recibe la información de control desde la CPU. Cargando los bits adecuados en este registro, la interfaz y el dispositivo se pueden colocar en diferentes modos de operación. Por ejemplo, el puerto A se puede definir como puerto solo de entrada. A una unidad de cinta se le puede mandar que rebobine la cinta o que empiece a adelantar la cinta. Los bits del registro de status se utilizan para las condiciones de status y para recoger errores que pueden ocurrir en la transferencia de datos. Por ejemplo, un bit de status puede indicar que el puerto A ha recibido un nuevo dato del dispositivo, mientras que otro bit del registro de status puede indicar que ha ocurrido un error de paridad durante la transferencia.

Los registros de la interfaz se comunican con la CPU a través de un bus de datos bidireccional. El bus de direcciones selecciona la unidad de interfaz a través de la entrada de selección de chip (*chip select*, CS) y las dos entradas de selección de registro. Un circuito (normalmente un decodificador o una puerta) detecta la dirección asignada a los registros de la interfaz. Este circuito habilita la entrada de selección del chip (CS) cuando se selecciona la interfaz mediante una dirección del bus. Las dos *entradas de selección de registro*, *RS*1 y *RS*0, se conectan normalmente a las dos líneas menos significativas del bus de direcciones. Las dos entradas selec-



CS	RS1	RS0	Registro seleccionado
0	x	x	Ninguno: bus de datos en alta impedancia
1	0	0	Registro puerto A
1	0	1	Registro puerto B
1	1	0	Registro de control
1	1	1	Registro de status

□ **FIGURA 13-5**
Ejemplo de una unidad de interfaz de E/S

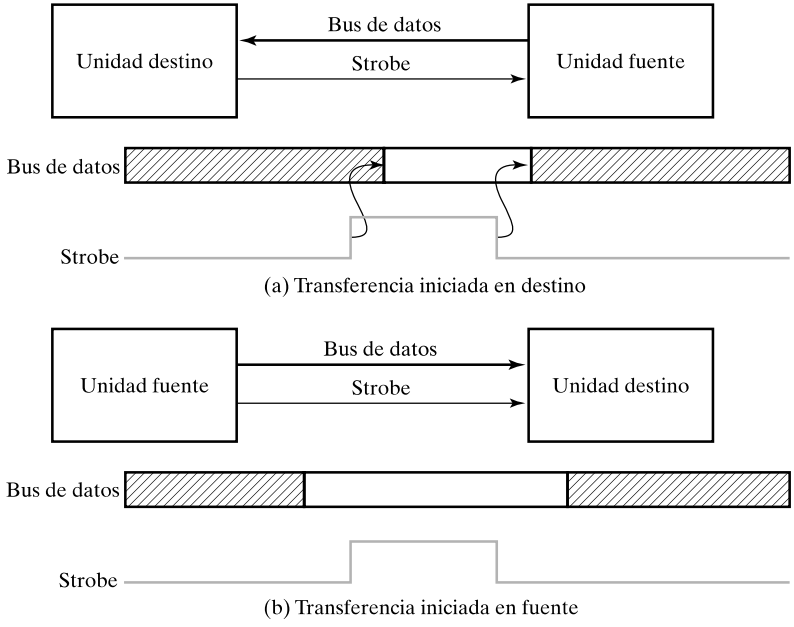
cionan uno de los cuatro registros del interfaz, según se especifica en la tabla que acompaña al diagrama de la Figura 13-5. El contenido del registro seleccionado se transfiere a la CPU a través del bus de datos cuando la señal de lectura de E/S se habilita. La CPU transfiere información binaria en el registro seleccionado a través del bus de datos cuando se habilita la señal de escritura de E/S.

La CPU, la interfaz y el dispositivo de E/S tienen, probablemente diferentes relojes que no están sincronizados unos con otros. Por esto se dice que estas unidades son *asíncronas* unas con respecto de las otras. Las transferencias asíncronas de datos entre dos unidades independientes necesitan que se transmitan señales de control entre las unidades para indicar el momento en el que dicho dato se empieza a transmitir. En el caso de la comunicación de la CPU a la interfaz, las señales de control deben indicar también el momento en el que la dirección es válida. Veremos dos métodos para realizar esta sincronización: *strobing* y *handshaking*. Inicialmente consideraremos casos generales en los que no hay direcciones involucradas, seguidamente añadiremos direccionamiento. A las unidades de comunicación, para el caso genérico, se les denominarán como unidad fuente y unidad destino.

Strobing

En la Figura 13-6 se muestra la transferencia de datos utilizando el método de *strobing*[1]. Se supone que el bus de datos entre las dos unidades se ha hecho bidireccional mediante el uso de buffers triestado.

[1] *N. del T.: Strobing* podría traducirse aquí como habilitación o petición.



□ **FIGURA 13-6**
Transferencia asíncrona utilizando *Strobing*

La transferencia de la Figura 13-6(a) se inicia por la unidad de destino. En la zona sombreada de la señal de datos, el dato no es válido. Además, un cambio en la señal *Strobe* (en la cola de cada flecha) provoca un cambio en el bus de datos (en la punta de la flecha). La unidad de destino cambia la señal *Strobe* de 0 a 1. Cuando el valor 1 de *Strobe* alcanza la unidad fuente, la unidad responde reemplazando el dato del bus de datos. La unidad destino espera que el dato esté disponible, en el peor de los casos, una cantidad de tiempo fija desde que *Strobe* haya pasado a 1. En ese instante, la unidad de captura el dato en un registro y cambia *Strobe* de 1 a 0. En respuesta al valor 0 de *Strobe*, la unidad fuente retira el dato del bus de datos.

La transferencia de la Figura 13-6(b) se inicia por la unidad fuente. En este caso, la unidad fuente coloca el dato sobre el bus de datos. Después de un corto espacio de tiempo necesario para colocar el dato en el bus, la unidad fuente cambia la señal Strobe de 0 a 1. En respuesta a Strobe igual a 1, la unidad de destino actualiza la transferencia a uno de sus registros. La fuente cambia luego Strobe de 1 a 0, lo que dispara la transferencia en el registro de destino. Finalmente, después de un corto periodo, para asegurar que la transferencia al registro se ha efectuado, la fuente quita el dato del bus de datos, completando así la transferencia.

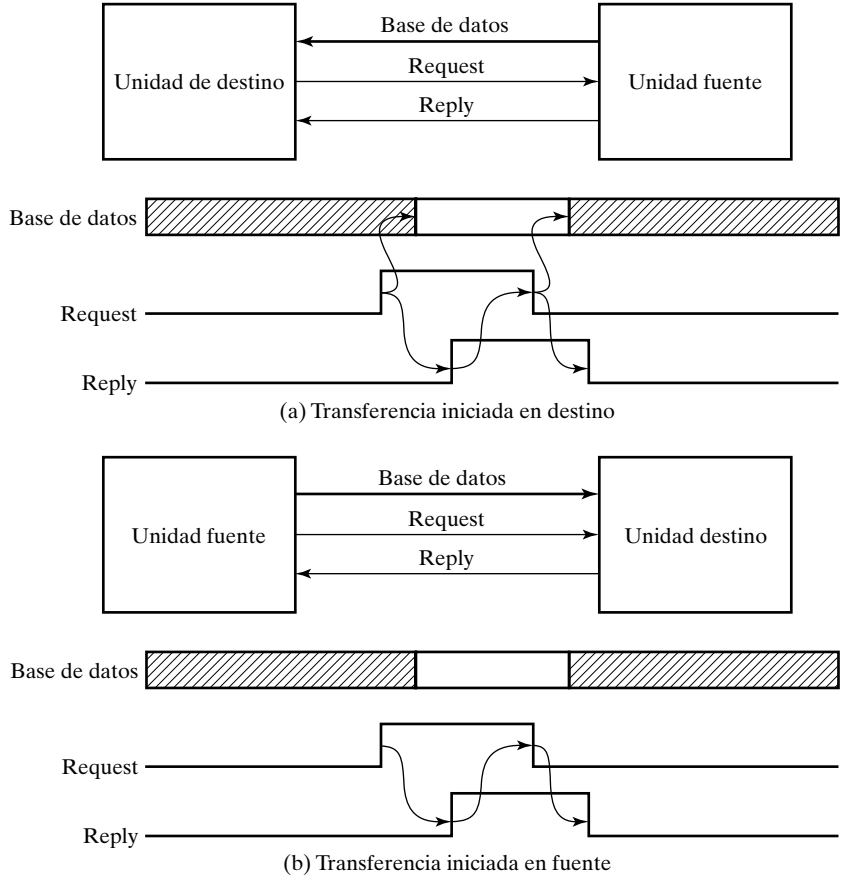
Aunque sencillo, el método de *strobing* para transferir datos tiene varias desventajas. Primero, cuando la unidad fuente inicia la transferencia, no se le indica a ésta que el dato no fue capturado por la unidad de destino. Es posible que, debido a un fallo en el hardware, la unidad de destino no haya recibido el cambio en la señal Strobe. Segundo, cuando la unidad de destino realiza la transferencia, no se le indica a ésta que la unidad fuente ha colocado realmente un dato en el bus. De esta forma, la unidad de destino podría leer valores arbitrarios del bus en lugar del valor real. Para terminar, la velocidad a la que las diferentes unidades responden pueden variar. Si hay varias unidades, la unidad que inicia la transferencia debe esperar, al menos, el tiempo de la unidad más lenta en la comunicación antes de cambiar la señal *Strobe* a 0. De esta forma, el tiempo que se usa en cada transferencia viene determinado por la unidad más lenta con la que una determinada unidad inicia las transferencias.

Handshaking

El método *handshaking*[2] utiliza dos señales de control para dirigir la temporización de la transferencia. Aparte de la señal desde la que una unidad inicia la transferencia, hay una segunda señal de control que procede de la otra unidad involucrada en la transferencia.

El principio básico del procedimiento de *handshaking* con dos señales para la transferencia de datos es como sigue. Una línea de control de la unidad que inicia la comunicación se usa para hacer una petición (*request*) de respuesta de la otra unidad. La segunda línea de control de la otra unidad se utiliza para responder a la unidad que inició la comunicación que la respuesta está efectuándose. De esta forma, cada unidad informa a la otra de su status y el resultado es una correcta transferencia a través del bus.

La Figura 13-7 muestra el procedimiento de transferencia de datos utilizando *handshaking*. En la Figura 13-7(a), la transferencia se inicia por la unidad de destino. Las dos líneas del protocolo se llaman *Request* (petición) y *Reply* (respuesta). El estado inicial es cuando tanto *Request* como *Reply* están deshabilitados y en el estado 00. Los siguientes estados son 10, 11 y 01. La unidad de destino inicia la transferencia habilitando *Request*. La unidad fuente responde colo-



□ **FIGURA 13-7**
Transferencia asíncrona utilizando *Handshaking*

[2] *N. del T.:* Handshaking podría traducirse aquí como protocolo de dos señales.

cando el dato en el bus. Después de un corto periodo de tiempo para mandar los datos por el bus, la unidad fuente activa *Reply* para indicar la presencia del dato. Como respuesta a *Reply*, la unidad de destino captura el dato en un registro y deshabilita la señal *Request*. La unidad fuente deshabilita entonces la señal *Reply* y el sistema va al estado inicial. La unidad de destino puede no realizar otra petición hasta que la unidad fuente haya mostrado su disponibilidad para proporcionar otro dato nuevo deshabilitando *Reply*. La Figura 13-7(b) representa el protocolo para la transferencia iniciada por la unidad fuente. En este caso, la fuente controla el intervalo entre cuando el dato se pone y *Request* cambia a 1, y entre que *Request* cambia a 0 y cuando el dato se quita.

El método de *handshaking* proporciona un alto grado de flexibilidad y fiabilidad debido a que la exitosa terminación de la transferencia cuenta con la participación activa de ambas unidades. Si una unidad tiene un fallo, la transferencia de datos no se completará. Este tipo de errores se pueden detectar mediante un mecanismo de *time-out*, que producen una alarma si la transferencia no se ha completado dentro de un intervalo de tiempo predeterminado. El *time-out* se realiza mediante un reloj interno que comienza la cuenta de tiempo cuando la unidad habilita una de sus señales de control. Si no se responde a las señales dentro de un periodo de tiempo determinado, la unidad supone que ha ocurrido un error. La señal de *time-out* se puede utilizar para interrumpir a la CPU y ejecutar una rutina de servicio que realice las acciones oportunas para solucionar el error. Además, el tiempo se controla en ambas unidades y no sólo en la unidad de inicio. Dentro de los límites del *time-out*, la respuesta de cada unidad a un cambio en una señal de control de la otra unidad puede llevar una cantidad arbitraria de tiempo y, aun así, la transferencia se realizará con éxito.

Los ejemplos de transferencias de las Figura 13-6 y 13-7 representan transferencias entre una interfaz y un dispositivo de E/S y entre una CPU y una interfaz. En el último caso, sin embargo, podría ser necesario el uso de una dirección para seleccionar la interfaz con la que la CPU desea comunicarse y un registro dentro de la interfaz. Para asegurar que la CPU direcciona la interfaz correcta, la dirección se debe enviar al bus de direcciones antes de que las señales de *Strobe* o de *Request* cambien de 0 a 1. Además, la dirección debe permanecer estable hasta que el cambio en las señales de *Strobe* o *Request* de 1 a 0 haya establecido un 0 en la interfaz lógica. Si se viola alguna de estas condiciones, se puede activar erróneamente otra interfaz, provocando una transferencia de datos incorrecta.

13-4 COMUNICACIÓN SERIE

La transferencia de datos entre dos unidades puede realizarse en serie o en paralelo. En las transferencias en paralelo, cada bit del mensaje tiene su propio camino y el mensaje completo se transmite de una vez. Esto quiere decir que un mensaje de *n* bits se transmite en paralelo mediante *n* caminos conductores separados. En una transmisión serie, cada bit del mensaje se envía en secuencia, de uno en uno. Este método necesita utilizar una o dos líneas para las señales. La transmisión en paralelo es más rápida pero necesita muchos hilos. Se usa en distancias cortas y cuando la velocidad es importante. La transmisión serie es más lenta pero menos cara puesto que solamente necesita un solo conductor.

Una forma en que las computadora y los terminales que están alejados unos de otros se conectan es mediante líneas telefónicas. Ya que las líneas telefónicas fueron originalmente diseñadas para comunicación de voz y las computadoras se comunican en base a señales digitales, es necesaria alguna forma de conversión. Los dispositivos que realizan esta conversión se llaman *módems* (del inglés *modulator-demodulators*). Un módem convierte señales digitales en tonos de audio que se transmiten por las líneas telefónicas y también convierten tonos de audio de la

línea telefónica en señales digitales para ser usadas por una computadora. Existen varios métodos de modulación, así como diferentes grados en los medios de comunicación y velocidades de transmisión. Los datos en serie se pueden transmitir entre dos puntos de tres modos: simplex, semi-duplex y duplex. Una línea *simplex* transporta información en una sola dirección. Este modo se utiliza raramente en comunicación de datos ya que el receptor no se puede comunicar con el transmisor para indicar si han ocurrido errores en la transmisión. Un ejemplo de transmisión *simplex* son las transmisiones de radio y televisión.

Un sistema de transmisión *semi-duplex* es aquel que es capaz de transmitir en ambas direcciones pero sólo en una dirección cada vez. Se necesitan un par de hilos para este modo. Una situación frecuente es la de un módem que actúa como transmisor y el otro como receptor. Cuando se completa la transmisión en una dirección, los papeles de los módems se cambian para permitir la transmisión en la dirección contraria. Al tiempo necesario para conmutar una línea semi-duplex de una dirección a otra se le llama *tiempo de respuesta*.

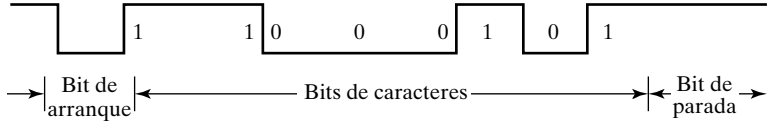
Un sistema de transmisión *duplex* puede recibir y enviar datos en ambas direcciones simultáneamente. Esto se puede conseguir con dos hilos más una conexión de tierra, con un hilo dedicado a cada dirección de transmisión. Alternativamente, un circuito de un solo hilo puede soportar comunicaciones *duplex* si el espectro de frecuencia se subdivide en dos bandas de frecuencias no solapadas para crear dos canales diferentes para transmitir y recibir en el mismo par de hilos.

La transmisión serie de datos puede ser síncrona o asíncrona. En las *transmisiones síncronas*, las dos unidades comparten una señal de reloj común y los bits se transmiten continuamente a esa frecuencia. En transmisiones serie de larga distancia, el transmisor y el receptor están gobernados por relojes distintos de la misma frecuencia. Se transmiten periódicamente unas señales de sincronización entre las dos unidades para mantener sus frecuencias de reloj en fase una con otra. En las *transmisiones asíncronas*, la información binaria sólo se manda cuando está disponible y las líneas están inactivas si no hay información a transmitir. Esto contrasta con la transmisión síncrona, en la que los bits se deben transmitir continuamente para mantener las frecuencias de reloj de ambas unidades sincronizadas.

Transmisión asíncrona

Una de las aplicaciones más comunes de la transmisión serie es la comunicación de una computadora con otro mediante módems conectados a través de la red telefónica. Cada carácter está formado por un código alfanumérico de ocho bits al que se le añaden bits adicionales al comienzo y al final del código. En las transmisiones asíncronas, cada carácter está formado por tres partes: el bit de arranque, los bits del carácter y el bit de parada. El convenio es que el transmisor mantiene la línea a 1 cuando no se transmiten caracteres. El primer bit, llamado bit de arranque es siempre 0 y se usa para indicar el comienzo del carácter. Un ejemplo de este formato se muestra en la Figura 13-8.

El receptor puede detectar un carácter transmitido aplicando las reglas de transmisión. Cuando no se envía un carácter la línea permanece en el estado 1. La inicialización de la transmisión



□ **FIGURA 13-8**
Formato del dato en una transmisión serie asíncrona

se detecta mediante el bit de arranque, que es siempre 0. Los bits del carácter siempre siguen al bit de arranque. Después de que el último bit del carácter se ha transmitido, se detecta un bit de parada cuando la línea vuelve a 1 durante al menos el tiempo necesario para transmitir un bit. Mediante estas reglas, el receptor puede detectar el bit de arranque cuando la línea pasa de 1 a 0. Mediante el uso de un reloj, el receptor examina la línea en los instantes de tiempo adecuados para determinar el valor del bit. El receptor conoce la tasa de transferencia de los bits y el número de caracteres por bit que acepta.

Después de que los bits del carácter se han transmitido, se mandan uno o dos bits de parada. Los bits de parada son siempre 1 e indican el final del carácter para señalizar el estado de espera o de inactividad. Estos bits permiten resincronizarse al transmisor y al receptor. La cantidad de tiempo que la línea permanece a 1 depende de la cantidad de tiempo necesario para resicronizar a los equipos. Algunos terminales electromecánicos antiguos utilizan dos bits de parada pero los equipos modernos sólo utilizan uno. La línea permanece a 1 hasta que se transmite otro carácter. El tiempo de parada asegura que no se transmitirá un nuevo carácter durante el tiempo correspondiente a transmitir uno o dos bits.

Como ejemplo, considere una transmisión serie con una tasa de transferencia de 10 caracteres por segundo. Suponga que cada carácter transmitido está formado por un bit de arranque, 8 bits por carácter y 2 bits de parada, es decir, un total de 11 bits. Si los bits se transmite a una velocidad de 10 bits por segundo, cada bit necesita 0.1 segundo para transferirse, como se van a transmitir 11 bits, se deduce que el *tiempo de bit* es de 9.09 ms. La *tasa de baudios* se define como el número máximo de cambios por segundo en la señal que esta siendo transmitida. Diez caracteres por segundo con un formato de 11 bits tienen una tasa de transferencia de 110 baudios.

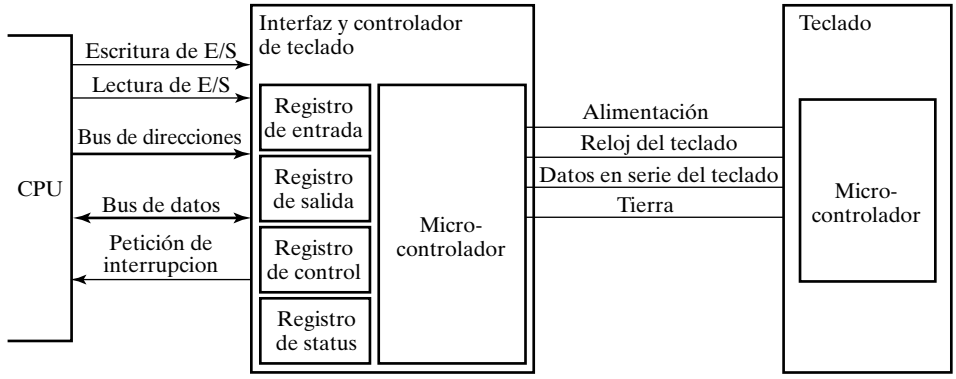
Transmisión síncrona

La transmisión síncrona no utiliza ni bit de arranque ni bits de parada para delimitar los caracteres. Los módems empleados en la transmisión asíncrona tienen relojes internos que se cambian a la frecuencia a la que se van a transmitir los bits. Para operar adecuadamente, se necesita que los relojes del módem del transmisor y del receptor permanezcan sincronizados todo el tiempo. Sin embargo, la línea de comunicación solo transporta los bits del dato, de cuya información se debe extraer la frecuencia de reloj. La frecuencia de sincronización se proporciona al módem receptor a partir de las transiciones de la señal del dato que se recibe. Cualquier desplazamiento en la frecuencia que pueda ocurrir entre los relojes del transmisor y el receptor se ajusta continuamente manteniendo el reloj del receptor a la misma frecuencia de la cadena de bits que llega. De esta forma se mantiene la misma tasa de transferencia tanto en el transmisor como en el receptor.

Contrariamente a lo que sucede en la transmisión asíncrona, en la que cada carácter puede ser enviado separadamente con sus propios bits de arranque y de parada, la transmisión síncrona debe mandar un mensaje continuo con el objeto de mantener el sincronismo. El mensaje está formado por un conjunto de bits que forman un bloque de datos. El bloque completo se transmite con bits de control especiales al comienzo y al final para delimitar el bloque en una unidad de información.

De vuelta al teclado

Hasta este punto hemos cubierto la naturaleza básica de la interfaz de E/S y la transmisión serie. Con estos dos conceptos vistos, estamos preparados ahora para continuar con el ejemplo del teclado y su interfaz, como se muestra en la Figura 13-9. El código de *scan-K*, producido por el



□ **FIGURA 13-9**
Controlador de teclado y su interfaz

microcontrolador del teclado, se ha de transferir vía serie desde el teclado, mediante su cable, al controlador del teclado de la computadora. La transferencia en la línea serie de teclado utiliza un formato como el mostrado para la transferencia asíncrona de la Figura 13-8. Sin embargo, en este caso, también se manda una señal de reloj del teclado a través del cable. De esta forma, la transmisión es síncrona con la señal de reloj transmitida, en lugar de asíncrona. Estas mismas señales se utilizan para transmitir comandos de control al teclado. En el controlador de teclado, el microcontrolador convierte el código *scan-K* a un *código de rastreo* más estándar, que se coloca posteriormente en el Registro de Entrada, a la vez que se manda una señal de interrupción a la CPU indicando que se ha pulsado una tecla y hay un código disponible. La rutina de atención a la interrupción lee el código de rastreo del registro de entrada y lo lleva a un área especial de la memoria. Esta área se manipula mediante el software almacenado en el Sistema Básico de Entrada/Salida (en inglés *Basic Input/Output System*, BIOS) que puede traducir el código de rastreo en un código de carácter ASCII para usar en las aplicaciones.

El Registro de Salida de la interfaz recibe los datos de la CPU. Los datos pueden pasarse al control de teclado, por ejemplo, cambiando la tasa de repetición cuando una tecla se mantiene pulsada. El Registro de Control se utiliza para los comandos del controlador del teclado. Finalmente, el Registro de Status da información específica sobre el status del teclado y del controlador del teclado.

Quizás uno de los aspectos más interesantes de la E/S del teclado es su alta complejidad. Involucra a dos microcontroladores que ejecutan diferentes programas, más el procesador principal que ejecuta el software de la BIOS (es decir, hay tres procesadores diferentes que ejecutan diferentes programas).

Un bus de E/S serie basado en paquetes

La E/S serie, como se describió para el teclado, utiliza un cable serie específicamente dedicado a la comunicación entre la computadora y el teclado. Ya sea paralelo o serie, hay conexiones de E/S externas típicamente dedicadas. El uso de estas rutas dedicadas necesitan con frecuencia que la caja de la computadora sea abierta e insertar tarjetas con circuitos electrónicos y conectores específicos para una E/S estándar usada para un dispositivo de E/S.

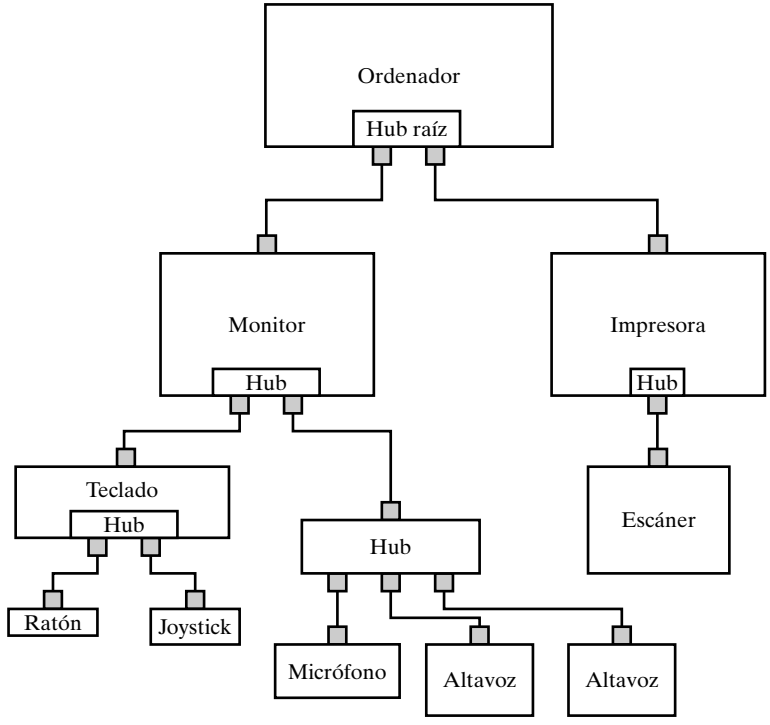
Por el contrario, la E/S serie basada en paquetes permite que muchos dispositivos de E/S diferentes utilicen una estructura de comunicación compartida que se conecta la computadora a

través de uno o dos conectores. El tipo de dispositivos soportados incluye a los teclados, ratones, *joysticks*, impresoras, scanners y altavoces. La entrada serie basada en paquetes concreta que describiremos aquí es el Bus Serie Universal (*Universal Serial Bus*, USB), que se está convirtiendo en una elección muy popular como forma de conexión para dispositivos de E/S de baja y media velocidad.

La interconexión de los dispositivos de E/S mediante USB se muestra en la Figura 13-10. La computadora y los dispositivos conectados a éste se pueden clasificar como *hubs* (bocas de conexión), dispositivos o componentes de dispositivos. Un *hub* proporciona el punto de conexión para los dispositivos USB y para otros *hubs*. Un *hub* contiene una interfaz USB para el manejo del control y status, y un repetidor para transferir información a través del *hub*.

La computadora contiene un controlador USB y el *hub* raíz (*root hub*). Otros *hubs* adicionales pueden formar parte de la estructura de la E/S USB. Si un *hub* se combina con un dispositivo como el teclado de la Figura 13-10, entonces al teclado se le denomina como dispositivo componente. Aparte de tales dispositivos componentes, un dispositivo USB sólo contiene un puerto USB para sacar sólo su función. El escáner es un ejemplo de un dispositivo USB habitual. Sin USB, el monitor, teclado, *mouse*, *joystick*, micrófono, altavoces, impresora y scanner mostrados deberían tener conexiones diferentes de E/S directamente conectadas en la computadora. El monitor, impresora, escáner, micrófono y los altavoces pueden necesitar tarjetas especiales que se han de insertar en la computadora, según se discutió anteriormente. Con USB sólo se necesitan dos conexiones.

Los cables USB contienen cuatro hilos: tierra, alimentación y dos líneas de datos (D+ y D−) utilizadas para señalización diferencial. El hilo de alimentación se utiliza para proporcio-



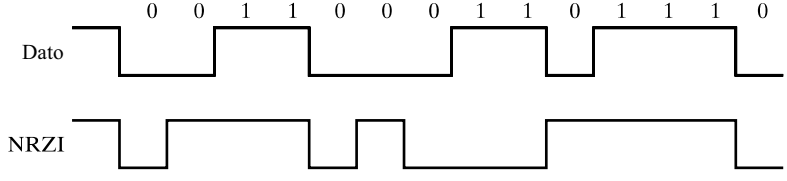
□ **FIGURA 13-10**
Conexión de dispositivos de E/S utilizando el Bus Serie Universal (USB)

nar pequeñas potencias a dispositivos como los teclados, de forma que no necesitan tener su propia alimentación. Para proporcionar inmunidad a las variaciones de la señal y al ruido, los 0 y 1 se transmiten utilizando la diferencia en voltios entre D+ y D−. Si la tensión en D+ excede la tensión en D− en 200 milivoltios o más, el valor lógico es 0. Otras relaciones de voltajes entre D+ y D− se utilizan también como estados especiales de la señal.

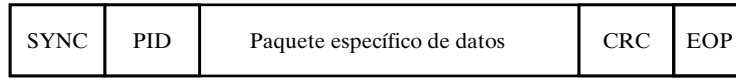
Los valores lógicos utilizados para la señalización no son valores lógicos de información que se transmiten. En su lugar, se utiliza una convención de señalización de no retorno a cero invertida (*Non-Return-to-Zero Inverted*, NRZI). Un cero del dato transmitido se representa mediante una transición de 1 a 0 o de 0 a 1, y un uno se representa mediante valores fijos de 1 o 0. La relación entre el dato que se transmite y la representación en NRZI se ilustra en la Figura 13-11. Como es típico en los dispositivos de E/S, no hay un reloj común para la computadora y el dispositivo. La codificación NRZI proporciona flancos que se pueden usar para mantener la sincronización entre los datos que llegan y el tiempo en el que cada bit se debe muestrear en el receptor. Si hay un número grande de 1s en serie en el dato, no habrá transiciones durante algún tiempo en el código NRZI. Para prevenir la pérdida de sincronización, se coloca un cero antes cada posición séptima de bit en una cadena de 1s antes de codificar a NRZI de forma que no aparecen más de seis 1s en la serie. El receptor debe ser capaz de eliminar estos ceros extra cuando convierta los datos NRZI en datos normales.

La información USB se transmite en paquetes. Cada paquete contiene un conjunto específico de campo dependiendo del tipo de paquete. Las cadenas lógicas de paquetes se usan para formar las operaciones USB. Por ejemplo, una operación de salida está formada por un paquete Salida seguido de un paquete Dato y un paquete de Protocolo. El paquete Salida procede del controlador USB de la computadora y notifica al dispositivo que está preparado para recibir un dato. La computadora manda luego el paquete Dato. Si el paquete se recibe sin error, el dispositivo responde con un paquete de Protocolo de reconocimiento (*Acknowledge Handshake packet*). Luego detallaremos la información contenida en cada uno de estos paquetes. La Figura 13-12(a) muestra un formato general para los paquetes USB y los formatos para cada uno de los paquetes involucrados en una transacción de salida. Nótese que cada paquete comienza con el patrón de sincronización SYNC. Este patrón es 00000001. Debido a la secuencia de ceros, el patrón correspondiente en NRZI contiene siete flancos que proporcionan un patrón para que el reloj de recepción se pueda sincronizar. Como este patrón se recibe mediante un estado especifico de una señal de tensión llamada IDLE, el patrón también señala el comienzo de un nuevo paquete.

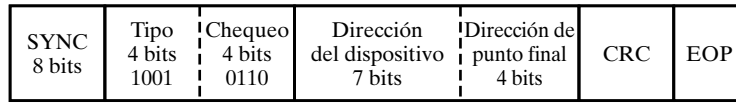
Siguiendo a SYNC, cada uno de los formatos de los paquetes contiene 8 bits denominados identificadores de paquete (PID). En el PID se especifica el tipo de paquete mediante 4 bits, con 4 bits adicionales que son el complemento de los 4 primeros para proporcionar un chequeo de error de tipo. Se detectarán una gran variedad de tipos de errores mediante la repetición del tipo con su complemento. El tipo va seguido opcionalmente por información específica del paquete, que varía dependiendo del tipo de paquete. Opcionalmente puede aparecer luego un campo de CRC. El patrón CRC, formado por 5 o 16 bits, es un patrón de Chequeo de Redundancia



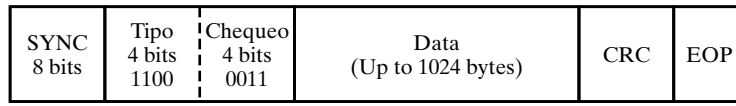
□ **FIGURA 13-11**
Representación de datos NRZI



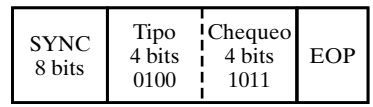
(a) Formato de paquete general



(b) Paquete de salida



(c) Paquete de datos (Tipo Data0)



(d) Paquete de protocolo (Tipo reconocimiento)

□ **FIGURA 13-12**
Formatos de los paquetes USB

Cíclica. Este patrón se calcula en la transmisión del paquete a partir de los datos específicos del paquete. El mismo cálculo se realiza cuando se recibe el dato. Si el patrón CRC no coincide con el patrón nuevamente calculado es que se ha detectado un error. Como respuesta al error, se puede ignorar el paquete y retransmitirlo. En el último campo del paquete aparece un Fin de Paquete (*End of Packet*, EOP). Este está formado por D+ y D−, ambos a nivel bajo durante dos tiempos de bit. Como su nombre indica, esta secuencia de señales de estado identifica el fin del paquete en curso. Se debe notar que en todos los campos se presenta primero el bit menos significativo.

En referencia a la Figura 13-12(b), en el paquete de salida (*Output Packet*), los campos Tipo y Chequeo van seguidos de una dirección de dispositivo (*Device Address*), una dirección de punto final (*Endpoint Address*) y un patrón CRC. La dirección de dispositivo está formada por siete bits y define el dispositivo de entrada de datos. La dirección de punto final está formada por cuatro bits y define por qué puerto del dispositivo se recibe la información en el siguiente paquete de datos. Por ejemplo, puede haber un puerto de datos y otro para el control de un determinado dispositivo.

Para el paquete de datos (*Data Packet*), sus datos específicos están formados entre 0 a 1024 bytes de datos. Debido a la longitud del paquete son más probables los errores compuestos, por eso se incrementa la longitud del patrón del CRC a 16 bits y mejorar la capacidad de detección. En el paquete de Protocolo (*Handshake Packet*), el paquete específico para datos está vacío. La respuesta a la recepción del paquete de datos se lleva mediante el PID. El PID 01001011 es un reconocimiento (ACK) que indica que el paquete se recibió sin detectar ningún error. La ausencia de paquetes de Protocolo cuando debería aparecer normalmente es una indicación de error. El PID 01011010 es un No Reconocimiento, indicando que el destino no está disponible temporalmente para aceptar o devolver datos. El PID 01111000 es una parada (STALL), indicando que el destino no está disponible para terminar la transferencia y que se necesita la intervención software para recuperarlo de la condición de parada.

Los anteriores conceptos ilustran los principios generales que subyacen en el bus de E/S basado en paquetes vía serie y que son específicos del USB. USB soporta otro tipo de paquetes y transacciones de muchos tipos diferentes. Además, la conexión y desconexión de los dispositivos se detectan y se pueden disparar diversas acciones de tipo software. En general, hay un software considerable en la computadora que soporta los detalles del control y operación del Bus Serie Universal.

13-5 MODOS DE TRANSFERENCIA

La información recibida de un dispositivo externo se almacena habitualmente en la memoria para un posterior procesado. La información transferida desde el procesador central a un dispositivo externo tiene su origen en la memoria. La CPU sólo ejecuta las instrucciones de E/S y puede aceptar datos temporalmente, pero la fuente y el destino último es la memoria. La transferencia de datos entre el procesador central y los dispositivos de E/S se puede manejar de varios modos, algunos de los cuales utiliza la CPU como camino intermedio, mientras que otros transfieren los datos directamente a o desde la memoria. La transferencia de datos a o desde los periféricos se puede manejar de cuatro modos posibles:

1. Transferencia de datos bajo control.
2. Transferencia de datos iniciada mediante interrupción
3. Transferencia con acceso directo a memoria
4. Transferencia mediante un procesador de E/S

Las operaciones controladas por programa son el resultado de las instrucciones de E/S escritas en el programa del procesador. Cada transferencia de datos se inicia mediante una instrucción del programa. Normalmente, la transferencia es entre un registro de la CPU y la memoria. La transferencia de datos bajo control de programa necesita monitorizar constantemente el periférico con la CPU. Una vez que se inicia la transferencia de datos, la CPU necesita monitorizar la interfaz para ver cuándo se hace de nuevo una transferencia. Las instrucciones programadas se ejecutan en la CPU para mantener una relación cercana con todo lo que está ocurriendo en la unidad de interfaz y en el dispositivo externo.

En la transferencia controlada por programa, la CPU permanece en un *bucle de espera* del programa hasta que la E/S indica que está preparada para la transferencia de datos. Esto es un proceso que consume tiempo ya que el procesador está ocupado innecesariamente. El bucle se puede evitar utilizando interrupciones y comandos especiales para informar a la interfaz que active una señal de petición de interrupción cuando el dato esté disponible en el dispositivo. Esto permite a la CPU ejecutar otros programas. La interfaz se mantiene mientras tanto monitorizando el dispositivo. Cuando la interfaz determina que el dispositivo está preparado para la transferencia de datos, ésta genera una petición de interrupción al procesador. Sobre la detección de la señal externa de interrupción, la CPU detiene momentáneamente la tarea que está realizando, se bifurca al programa de servicio para procesar la transferencia de datos y luego regresa a la tarea original que se interrumpió. Esta transferencia iniciada por interrupción es el tipo de transferencia que utiliza el controlador del teclado mostrado en la Figura 13-9.

La transferencia de datos bajo el control del programa se realiza a través del bus de E/S y entre la CPU y una unidad de interfaz periférica. En el acceso directo a memoria (*direct memory access*, DMA), la unidad de interfaz transfiere los datos dentro y fuera de la memoria mediante los buses de la memoria. La CPU inicia la transferencia proporcionando a la interfaz la dirección de comienzo y el número de palabras que se quieren transferir y luego seguir realizando

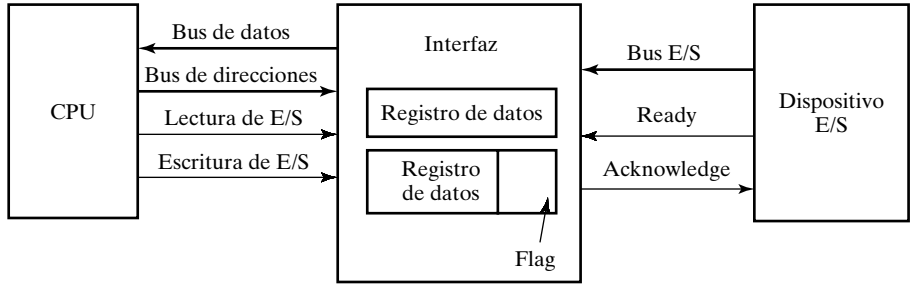
otras tareas. Cuando la transferencia se hace, la interfaz pide ciclos de memoria usando los buses de la memoria. Cuando se concede la petición al controlador de la memoria, la interfaz transfiere los datos directamente a la memoria. La CPU sólo retrasa las operaciones con la memoria para permitir la transferencia de E/S directa de la memoria. Como la velocidad de los periféricos es normalmente más lenta que la del procesador, las operaciones de transferencias de E/S de la memoria son infrecuentes comparadas con los accesos del procesador a la memoria. La transferencia con DMA se discute con más detalle en la Sección 13-7.

Muchos procesadores combinan la interfaz con los requisitos para el DMA en una unidad llamada procesador de E/S (*I/O processor*, IOP). El IOP puede manejar muchos periféricos mediante la combinación de DMA e interrupción. En dichos sistemas se divide al procesador en tres módulos diferente: la unidad de memoria, la CPU y el IOP. Los procesadores de E/S se presentan en la Sección 13-8

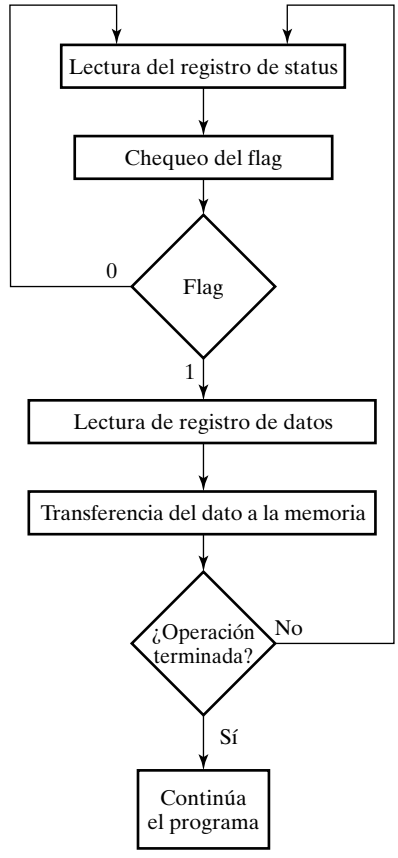
Ejemplo de una transferencia controlada por programa

En la Figura 13-13 se muestra un sencillo ejemplo de transferencia de datos desde un dispositivo de E/S a través de una interfaz en la CPU. El dispositivo transfiere bytes de datos, de uno en uno, según están disponibles. Cuando un byte está disponible, el dispositivo lo coloca en el bus de E/S y habilita la señal *Ready* (preparado). La interfaz acepta el byte en su registro de datos y habilita la señal *Acknowledge* (reconocimiento). La interfaz cambia un bit del registro de status, al que llamaremos *bandera* o *flag*. El dispositivo puede deshabilitar *Ready* ahora pero no se transferirá otro byte hasta que la interfaz deshabilite *Acknowledge*, de acuerdo con el procedimiento de *handshaking* establecido en la Sección 13-3.

Bajo el control del programa, la CPU debe chequear el *flag* para determinar si hay un nuevo byte en el registro de datos de la interfaz. Esto se hace leyendo el contenido del registro de status, llevándolo a un registro de la CPU, y chequear el valor del *flag*. Si el *flag* es igual a 1 la CPU lee el dato del registro de datos. Luego, la CPU o la interfaz ponen el *flag* a 0, dependiendo de cómo se haya diseñado la interfaz del circuito. Una vez que el *flag* se ha puesto a 0, la interfaz deshabilita la señal de *Acknowledge* y el dispositivo puede transferir el siguiente dato. En la Figura 13-14 se muestra el diagrama de flujo de un programa para realizar la anterior transferencia. El diagrama de flujo supone que el dispositivo está mandando una secuencia de bytes que debe almacenarse en la memoria. El programa examina continuamente el status de la interfaz hasta que el *flag* se pone a 1. Cada byte se introduce en la CPU y se introduce en la memoria hasta que todos los datos han sido transferidos.



□ **FIGURA 13-13**
Transferencia de datos de un dispositivo de E/S a la CPU



□ **FIGURA 13-14**
Diagrama de flujo para el programa de una CPU.

La transferencia de datos controlada por programa se usa sólo en sistemas que se dedican a monitorizar el dispositivo continuamente. La diferencia entre la tasa de transferencia de información de la CPU y el dispositivo de E/S hace que este tipo de transferencia sea ineficiente. Para ver la causa, consideremos un procesador típico que ejecuta un conjunto de instrucciones para leer el registro de status y chequear el *flag* cada 100 ns. Suponga que el dispositivo de entrada transfiere sus datos con una frecuencia media de 100 bytes por segundo. Esto es equivalente a un byte cada 10 000 μs, es decir, que la CPU chequeará el *flag* cada 100 000 veces entre cada transferencia. De esta forma, la CPU esta gastando el tiempo en chequear el *flag* en lugar de hacer alguna tarea útil.

Transferencia iniciada por interrupción

Una alternativa a la monitorización constante del *flag* por la CPU es dejar que la interfaz informe al procesador cuando haya un dato listo para transferir. Este modo de transferencia utiliza las interrupciones. Cuando la CPU está ejecutando un programa, éste no chequea el *flag*. Sin embargo, cuando el *flag* se activa, el procesador interrumpe momentáneamente el programa que está ejecutando y se le informa del hecho de que el *flag* ha sido activado. La CPU deja lo que estaba haciendo para ocuparse de la transferencia de entrada o de salida. Después de completar

la transferencia, el procesador regresa al programa anterior para continuar con lo que estaba haciendo antes de la interrupción. La CPU responde a la señal de interrupción almacenando la dirección de retorno del contador de programa en una memoria de pila o registro, y el control salta a la rutina de servicio que procesa la petición de transferencia de E/S. La forma en la que el procesado elige la dirección de bifurcación de la rutina de servicio varía de unas unidades a otras. En principio, hay dos métodos para acometer esto: *interrupciones vectorizadas* y *no vectorizadas*. En una interrupción no vectorizada, la dirección de ramificación se asigna a una posición fija de la memoria. En una interrupción vectorizada, la fuente que interrumpe proporciona la dirección de salto al procesador. A esta dirección se le llama *vector de dirección*. En algunos procesadores, el vector de dirección es la primera dirección de la rutina de servicio; en otros procesadores, el vector de dirección es una dirección que apunta a una posición de memoria donde está almacenada la primera dirección de la rutina de servicio. El procedimiento de interrupción vectorizada se presentó en la Sección 11-9 junto con la Figura 11-9.

13-6 PRIORIDAD EN LAS INTERRUPCIONES

Un procesador típico tiene conectados un determinado número de dispositivo de E/S que son capaces de originar una petición de interrupción. La primera tarea del sistema de interrupción es identificar la fuente de interrupción. Hay también la posibilidad de que varias fuentes de interrupción soliciten ser atendidas simultáneamente. En este caso, el sistema debe decidir a qué dispositivo atiende primero.

Un sistema de prioridad para las interrupciones establece una prioridad sobre varias fuentes de interrupción para determinar qué petición de interrupción atiende primero cuando dos o más de ellas llegan simultáneamente. El sistema puede determinar qué interrupciones tienen permiso para interrumpir al procesador mientras que otra interrupción está siendo atendida. Lo niveles más altos de prioridad se asignan a las peticiones que, si se retrasan o interrumpen, podrían tener serias consecuencias. Los dispositivos con alta velocidad de transferencia, como los discos magnéticos, tienen dada alta prioridad, y los dispositivos más lentos, como los teclados, reciben la prioridad más baja. Cuando dos dispositivos interrumpen al procesador al mismo tiempo, el procesador atiende al dispositivo con mayor prioridad primero.

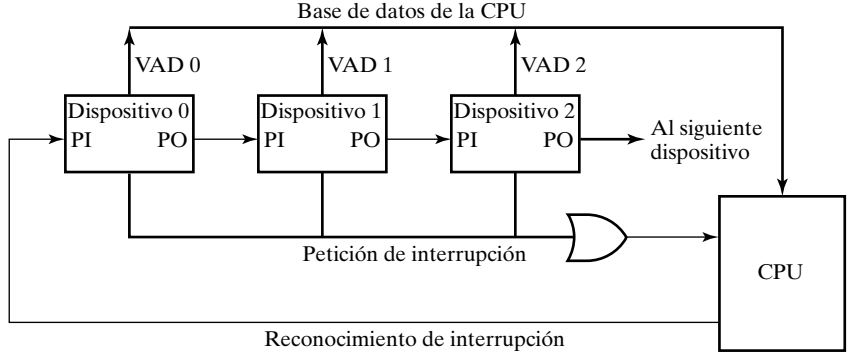
El establecimiento de la prioridad de interrupciones simultáneas se puede hacer por software o por hardware. El software utiliza un procedimiento de interrogación o sondeo (*polling*), para identificar la fuente de interrupción de mayor prioridad. En este método, hay una dirección común de salto para todas las interrupciones. El programa atiende a las interrupciones en la dirección de bifurcación haciendo una secuencia de sondeo a las fuentes de interrupción. La prioridad de cada fuente de interrupción determina el orden en que se van a interrogar, es decir, el orden del sondeo. La fuente con la prioridad más alta se comprueba primero, y si su señal de interrupción está activada, el control salta a la subrutina que atiende a dicha fuente. De otra manera, la fuente con la siguiente prioridad más baja se comprueba, y así sucesivamente. De esta forma, la rutina inicial de servicio para todas las interrupciones está compuesta de un programa que comprueba las fuentes de interrupción secuencialmente y salta a una de las posibles rutinas de atención. La primera rutina de servicio que se realiza pertenece al dispositivo de prioridad más alta de todos los que pueden interrumpir al procesador. La desventaja del método software es que si hay muchas interrupciones, el tiempo necesario para sondear a todas las fuentes puede exceder del tiempo disponible para atender a los dispositivos de E/S. En esta situación se puede usar una unidad de interrupción con prioridad hardware para acelerar el funcionamiento del sistema.

Una unidad de interrupción con prioridad hardware funciona como un gestor general para el entorno de un sistema de interrupciones. La unidad acepta las peticiones de interrupción de varias fuentes, determina cual de las peticiones entrantes tienen la prioridad más alta y pasa la petición de interrupción al procesador basándose en esa información. Para acelerar el funcionamiento, cada fuente de interrupción tiene su propio vector de interrupción para acceder a su propia rutina directamente. De esta forma, no se necesita hacer un sondeo porque todas las decisiones se han hecho en la unidad de interrupción con prioridad hardware. La función de prioridad hardware se puede establecer tanto para conexiones series o paralelas de las líneas de interrupción. La conexión serie es también conocida como *Daisy Chain*.

Prioridad *Daisy Chain*

El método *Daisy Chain* de establecimiento de prioridad consiste en una conexión serie de todos los dispositivos que solicitan una interrupción. El dispositivo con la prioridad más alta se coloca en la primera posición, seguido de los demás dispositivos en orden descendente de prioridad, hasta el dispositivo con prioridad más baja, que se coloca el último de la cadena. Este modo de conexión entre tres dispositivos y la CPU se muestra en la Figura 13-15. Las líneas de interrupción de todos los dispositivos se conectan a una puerta OR para formar una línea de interrupción, que va conectada a la CPU. Si un dispositivo tiene su petición de interrupción a 1, la línea de interrupción cambia a 1 y habilita la línea de entrada de interrupción de la CPU. Si no hay interrupciones pendientes, la línea de interrupción permanece a 0 y la CPU no reconoce ninguna interrupción. La CPU responde a una petición de interrupción habilitando la señal de reconocimiento de interrupción. La señal que se produce se recibe en el dispositivo 0 en su entrada PI (entrada de prioridad, *priority in*). La señal pasa al siguiente dispositivo a través de la salida PO (salida de prioridad, *priority out*) solo si el dispositivo 0 no está solicitando una interrupción. Si el dispositivo 0 tiene una interrupción pendiente, éste bloquea la señal de reconocimiento que va al siguiente dispositivo colocando un 0 en la salida PO y procede a introducir su dirección de vector de interrupción (VAD, del inglés *interrupt vector address*) en el bus de datos para que la CPU lo use durante el ciclo de interrupción.

Un dispositivo que tiene un 0 en su entrada PI genera un 0 en su salida PO, que informa al siguiente dispositivo con prioridad más baja que la señal de reconocimiento ha sido bloqueada. Un dispositivo que está solicitando una interrupción y tiene un 1 en su entrada PI interceptará la señal de reconocimiento colocando un 0 en su salida PO. Si el dispositivo no tiene ninguna



□ **FIGURA 13-15**
Método *Daisy Chain* de prioridad de interrupciones

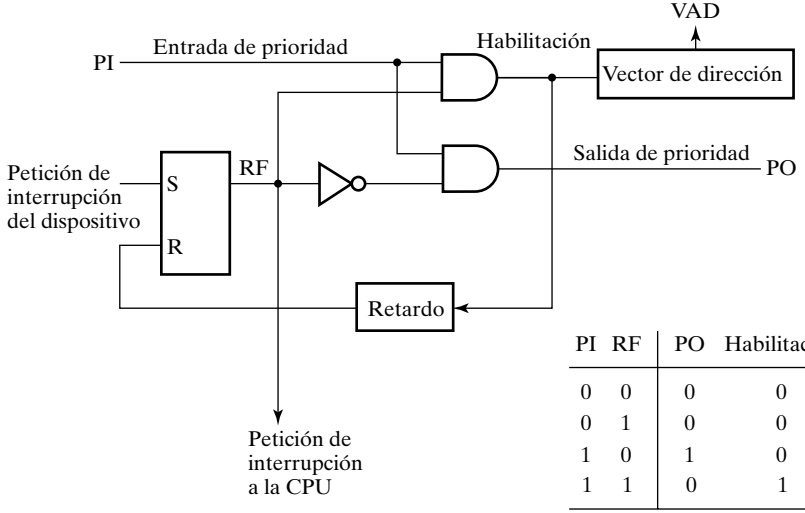
interrupción pendiente, transmite la señal de reconocimiento al siguiente dispositivo colocando un 1 en su salida PO. De esta forma, el dispositivo con PI = 1 y PO = 0 es el de mayor prioridad que está solicitando una interrupción, y este dispositivo es el que coloca su VAD en el bus de datos. La disposición de esta cadena, la *Daisy Chain*, da la mayor prioridad al dispositivo que recibe la señal de reconocimiento de la CPU. Cuanto más lejos esté el dispositivo de la primera posición, menor es su prioridad.

La Figura 13-16 muestra la lógica que se debe incluir dentro de cada dispositivo conectado en esta cadena. El dispositivo pone su latch RF a 1 cuando solicita una interrupción a la CPU. La salida del latch entra en la puerta OR que maneja la línea de interrupción. Si PI = 0, tanto PO como la línea de habilitación de VAD son iguales a 0, independientemente del valor de RF. Si PI = 1 y RF = 0, entonces PO = 0, el vector de dirección está deshabilitado y la señal de reconocimiento pasa al siguiente dispositivo a través de PO. El dispositivo está activo si PI = 1 y RF = 1, que coloca un 0 en PO y pone el vector de dirección en el bus de datos. Se supone que cada dispositivo tiene su propio vector de dirección, diferente unos de otros. El latch RF se pone a 0 después de un retardo lo suficientemente grande para asegurar que la CPU ha recibido el vector de dirección.

Hardware de prioridad paralela

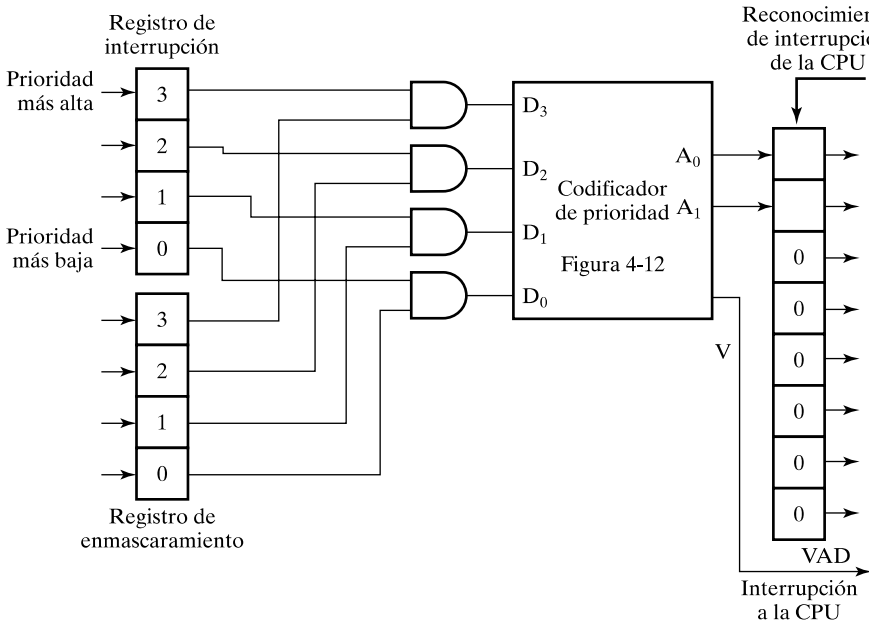
El método de prioridad paralela utiliza un registro con un conjunto de bits que cambian separadamente según la señal de interrupción de cada dispositivo. La prioridad se establece de acuerdo con la posición de los bits en el registro. Aparte del registro de interrupción, el circuito puede incluir un registro de enmascaramiento para controlar el status de cada petición de interrupción. El registro de enmascaramiento se puede programar para deshabilitar las interrupciones de prioridad más baja mientras que se atiende a un dispositivo de prioridad más alta. Esto también puede permitir que un dispositivo de alta prioridad interrumpa a la CPU cuando un dispositivo de prioridad más baja esta siendo atendido.

La lógica de prioridad de un sistema con cuatro fuentes de interrupción se muestra en la Figura 13-17. La lógica está formada por un registro de interrupción con un conjunto bits que



PI	RF	PO	Habilitación
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

□ **FIGURA 13-16**
Etapa de una cadena de prioridad *Daisy Chain*



□ **FIGURA 13-17**
Hardware para prioridad paralela

se ponen a 1 individualmente según ciertas condiciones externas y se ponen a 0 mediante las instrucciones del programa. La entrada de interrupción 3 tiene la prioridad más alta y la entrada 0 la más baja. El registro de enmascaramiento tiene el mismo número de bits que el registro de interrupciones. Mediante instrucciones de un programa es posible poner a 1 o a 0 cualquier bit del registro de enmascaramiento. Cada bit de interrupción y su bit de máscara correspondiente se aplica a una puerta AND para producir las cuatro entradas de un codificador de prioridad. De esta forma, se reconoce una interrupción sólo si su bit de máscara correspondiente se ha puesto a 1 mediante el programa. El codificador de prioridad genera dos bits del vector de direcciones que se transfieren a la CPU por el bus de datos. La salida *V* del codificador se pone a 1 si ha llegado una petición de interrupción que no está enmascarada. Esto proporciona la señal de interrupción para la CPU.

El codificador de prioridad es un circuito que realiza la función de prioridad. La lógica del codificador de prioridad es tal que, si dos o más entradas están activadas al mismo tiempo, la entrada que tiene la prioridad más alta tiene preferencia. En la Sección 4-4 se puede encontrar un codificador de prioridad de cuatro entradas, y su tabla de verdad se muestra en la Tabla 4-5. La entrada D_3 tiene la prioridad más alta, así que, independientemente de los valores de las otras entradas, cuando esta entrada es 1, la salida es $A_1\ A_0 = 11$. D_2 tiene la siguiente prioridad más baja. La salida es 10 si $D_2 = 1$, con tal de que $D_3 = 0$, independientemente de los valores de las dos entradas de prioridad más baja. La salida es 01 cuando $D_1 = 1$, siempre y cuando las dos entradas de prioridad más alta sean iguales a 0, y así sucesivamente bajando los niveles de prioridad. La salida de interrupción *V* es igual a 1 cuando una o más entradas son iguales a 1. Si todas las entradas son 0, *V* es 0, y las otras dos salidas del codificador no se utilizan. Esto es así porque el vector de dirección no se transfiere a la CPU cuando $V = 0$.

La salida del codificador con prioridad se utiliza para formar parte del vector de dirección de la fuente de interrupción. Los otros bits del vector de dirección pueden tener asignado cual-

quier valor. Por ejemplo, el vector de dirección se puede completar añadiendo seis ceros a las salidas del codificador. De esta forma, los vectores de interrupción de los cuatro dispositivos de E/S se asignan a los números de 8 bits equivalentes a 0, 1, 2 y 3.

13-7 ACCESO DIRECTO A MEMORIA

La transferencia de bloques de información entre un dispositivo rápido de almacenamiento, como un disco magnético, y una CPU puede ocupar a la CPU y permitirle hacer poco, si acaso, puede completar otra tarea. Eliminando la CPU de la ruta y dejando que el periférico gestione los buses de la memoria directamente, podemos relevar a la CPU de muchas operaciones de E/S y permitirle que prosiga con otras tareas. En esta técnica de transferencia, llamada acceso directo a memoria (*direct memory access*, DMA), el controlador de DMA se hace con los buses para gestionar la transferencia directa de información entre el dispositivo de E/S y la memoria. Como consecuencia, se priva temporalmente a la CPU del acceso a la memoria y del control de los buses de la memoria.

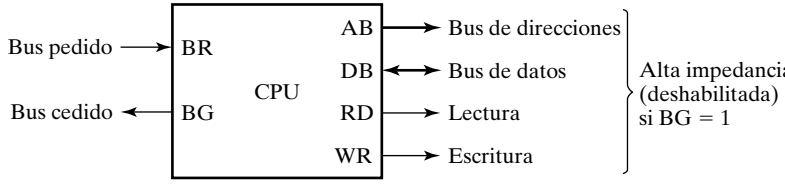
El DMA puede capturar los buses de diferentes formas. Un método usado comúnmente en los microprocesadores es deshabilitar los buses mediante señales de control especiales. La Figura 13-18 muestra dos señales de control de la CPU que facilita la transferencia de DMA. El controlador de DMA utiliza la señal de petición de bus (*bus request*, *BR*) para pedir a la CPU que ceda el control de los buses. Cuando la entrada *BR* se activa, la CPU coloca el bus de direcciones, el bus de datos y las líneas de lectura y escritura en estado de alta impedancia. Después de hacer esto, la CPU activa la señal de salida de bus concedido (*bus granted*, *BG*) para informar al DMA externo que puede tomar el control de los buses.

En todo el tiempo que la línea *BG* se mantiene activada, la CPU no tiene permiso para efectuar operaciones que necesiten acceder a los buses. Cuando la entrada de petición de bus está deshabilitada por el DMA, la CPU vuelve a su operación normal, deshabilita la señal de salida *BG* y toma el control de los buses.

Cuando la línea *BG* se habilita, el controlador externo de DMA toma el control de los buses del sistema para comunicarse directamente con la memoria. Se puede hacer una transferencia de palabras de memoria por bloques enteros, suspendiendo la operación de la CPU hasta que el bloque entero se transfiere, proceso que se denomina *ráfaga de transferencia*. O la transferencia se puede hacer palabra a palabra entre ejecuciones de instrucciones de la CPU, proceso llamado *transferencia de un solo ciclo* o *robo de ciclo*. La CPU tan sólo retrasa sus operaciones de bus un ciclo de memoria para permitir a la transferencia directa memoria-E/S robar un ciclo de memoria.

El controlador de DMA

El controlador de DMA necesita el habitual circuito de interfaz para comunicar la CPU y el dispositivo de E/S. Además necesita un registro de direcciones, un registro de cuenta de direc-



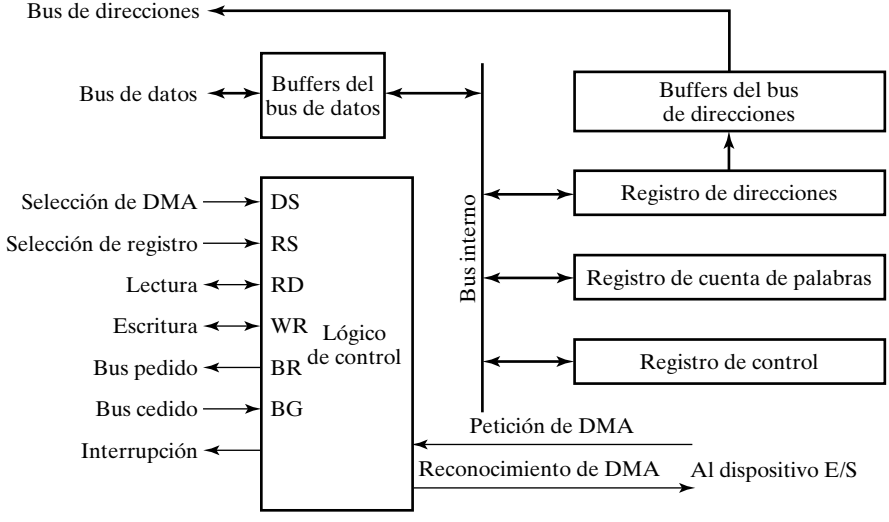
□ **FIGURA 13-18**
Señales de control del bus de la CPU.

ciones y un conjunto de líneas de dirección. El registro de dirección y las líneas de dirección se usan para dirigir la comunicación con la memoria. El registro de cuenta de direcciones especifica el número de palabras que se deben transferir. La transferencia de datos se puede hacer directamente entre el dispositivo y la memoria bajo el control del DMA.

La Figura 13-19 muestra el diagrama de bloques de un controlador de DMA típico. La unidad que se comunica con la CPU mediante el bus de datos y las líneas de control. La CPU selecciona los registros del DMA mediante el bus de direcciones, habilitando las entradas *DS* (*DMA select*) y *RS* (*Register select*). Las entradas *RD* (lectura) y *WR* (escritura) son bidireccionales. Cuando la entrada *BG* (bus cedido) es 0, la CPU se puede comunicar con los registros del DMA a través del bus de datos para leer o escribir en estos registros. Cuando $BG = 1$, la CPU ha cedido los buses y el DMA se puede comunicar directamente con la memoria especificando una dirección en el bus de direcciones y activando la línea de control *RD* o *WR*. El DMA se comunica con el periférico externo a través de las líneas de petición de DMA (*DMA request*) y de reconocimiento de DMA (*DMA acknowledge*), como se describió en el procedimiento de *handshaking*.

El controlador de DMA tiene tres registros: un registro de dirección, un registro de cuenta de palabras y un registro de control. El registro de dirección contiene una dirección para especificar la posición deseada de una palabra de la memoria. Los bits de direcciones van a través de los *buffers* del bus sobre el bus de direcciones. El registro de dirección se incrementa después de cada palabra que se ha transferido a la memoria. El registro de cuenta de palabras contiene el número de palabras a transferir. Este registro se decrementa en uno cada vez que se transfiere una palabra y se comprueba internamente si su contenido es cero. El registro de control especifica el modo de transferencia. Todos los registros en el DMA son para la CPU como registros de una interfaz de E/S. De esta forma, la CPU puede leer de o escribir en los registros del DMA bajo control de programa vía el bus de datos.

Después de la inicialización por la CPU, el DMA comienza y continúa transfiriendo datos entre la memoria y la unidad periférica hasta que el bloque entero se ha transferido. El proceso de inicialización es, esencialmente, un programa formado por instrucciones de E/S que incluyen



□ **FIGURA 13-19**
Diagrama de bloques de un controlador de DMA

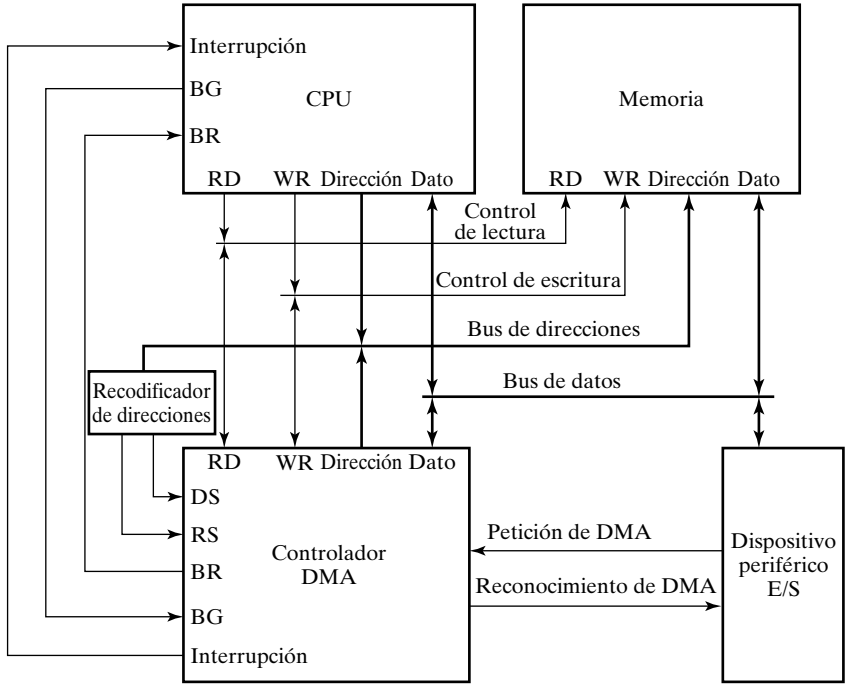
la dirección de un registro concreto del DMA. La CPU inicializa el DMA mandando la siguiente información por el bus de datos:

1. La dirección de comienzo del bloque de memoria en el que los datos están disponibles (para lectura) o van a ser almacenados (con escritura).
2. La cuenta de palabras, que es el número de palabras en el bloque de memoria.
3. Un bit de control que especifica el modo de transferencia, como lectura o escritura
4. Un bit de control para comenzar la transferencia de DMA

La dirección de comienzo se almacena en el registro de dirección, la cuenta de palabras en el registro de cuenta de palabras y la información de control en el registro de control. Una vez que el DMA se inicializa, la CPU detiene la comunicación con el DMA a menos que la CPU reciba una señal de interrupción o necesite chequear cuántas palabras se han transferido.

Transferencia de DMA

En la Figura 13-20 se ilustra la posición del controlador de DMA entre los otros componentes de un sistema con procesador. La CPU se comunica con el DMA a través del bus de direcciones y de datos como con cualquier unidad de interfaz. El DMA tiene su propia dirección que activa las líneas *DS* y *RS*. La CPU inicializa el DMA a través del bus de datos. Una vez que el DMA recibe el bit de control de inicio, puede comenzar a transferir datos entre el dispositivo periférico y la memoria. Cuando el periférico manda una petición de DMA, el controlador de DMA activa la línea *BR* informando a la CPU que está cediendo los buses. La CPU responde con su línea *BG*, informando al DMA de que los buses están deshabilitados. Entonces el DMA pone el



□ **FIGURA 13-20**
Transferencia de DMA en un sistema con procesador

valor actual de su registro de direcciones en el bus de direcciones, inicializando las señales *RD* y *WR* y manda un reconocimiento de DMA al periférico.

Cuando el periférico recibe el reconocimiento del DMA pone una palabra en el bus de datos (para escribir) o recibe una palabra del bus de datos (para leer). Así, el DMA controla la operación de lectura o escritura y aporta la dirección de memoria. La unidad periférica puede entonces comunicarse con la memoria a través del bus de datos para hacer una transferencia directa de datos entre las dos unidades, mientras que el acceso de la CPU al bus de datos está momentáneamente deshabilitado.

Para cada palabra que se transfiere, el DMA incrementa su registro de direcciones y decrementa su registro de cuenta de palabras. Si la cuenta de palabras no ha alcanzado el cero, el DMA comprueba la línea de petición procedente del periférico. En un dispositivo de alta velocidad, la línea se activará tan pronto como su anterior transferencia se haya terminado. Luego se inicia una segunda transferencia y el proceso continúa hasta que todo el bloque se haya transferido. Si la velocidad del periférico es baja, la línea de petición del DMA se puede activar algo más tarde. En este caso, el DMA deshabilita la línea de petición del bus de forma que la CPU puede continuar ejecutando su programa. Cuando el periférico solicita una transferencia, el DMA solicita los buses de nuevo.

Si el contador de palabras llega a cero, el DMA para cualquier transferencia posterior y elimina su petición de bus. Además informa a la CPU del fin de la transferencia mediante una interrupción. Cuando la CPU responde a la interrupción, lee el contenido del registro de cuenta de palabras. Si el valor es cero, esto indica que todas las palabras se han transferido con éxito. La CPU puede leer el registro de cuenta de palabras en cualquier momento, así como comprobar el número de palabras que ya se ha transferido.

Un controlador de DMA puede tener más de un canal. En este caso, cada canal tiene su pareja de señales de control de petición y reconocimiento, que se conectan a diferentes periféricos. Cada canal también tiene su propio registro de direcciones y su registro de cuenta de palabras y los canales con mayor prioridad se atienden antes que los canales de menor prioridad.

La transferencia de DMA es muy útil en muchas aplicaciones, incluyendo las transferencias rápidas entre discos magnéticos y la memoria y entre las tarjetas gráficas y la memoria.

13-8 PROCESADODES DE E/S

En lugar de tener cada interfaz comunicándose con la CPU, un procesador puede incorporar uno o más procesadores externos y asignarles luego la tarea de comunicarse directamente con todos los dispositivos de E/S. Un procesador de entrada/salida (IOP, del inglés *Input-Output Processor*) se puede clasificar como un procesador con capacidad de acceso directo a memoria que se comunica con los dispositivos de E/S. Con esta configuración, un sistema basado en procesador se puede dividir en una unidad de memoria y un número de procesadores compuestos por una CPU y uno o más IOPs. Cada IOP se encarga de las tareas de entrada y salida, relevando a la CPU de «los quehaceres domésticos» involucrados en las transferencias de E/S. A un procesador que se comunica con su unidad remota por teléfono u otros medios de comunicación vía serie se le llama *procesador de comunicación de datos* (DCP, del inglés *Data Communication Processor*). El beneficio derivado del uso de los procesadores de E/S es la mejora en el rendimiento del sistema, proporcionado mediante el relevo de la CPU de ciertas tareas relacionadas con la E/S y asignándoles los procesadores de E/S adecuados.

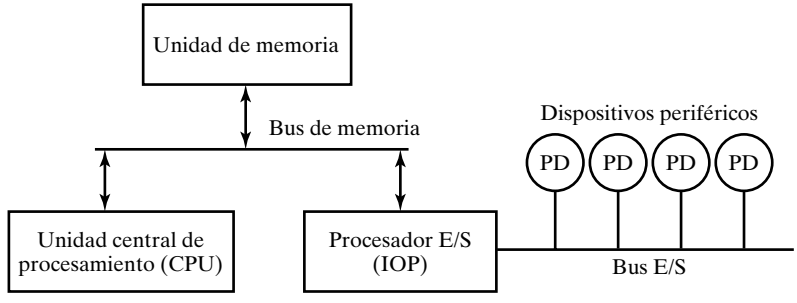
Un IOP es similar a una CPU, excepto que se diseña para manejar los detalles del procesamiento de la E/S. Al contrario que el controlador de DMA, que debe configurarse enteramente

por la CPU, el IOP puede acceder y ejecutar sus propias instrucciones. Las instrucciones del IOP se han diseñado específicamente para facilitar las transferencias de E/S. Además, el IOP puede realizar otras tareas de procesamiento como son aritméticas, lógicas, de bifurcación y traducción de código.

En la Figura 13-21 se muestra el diagrama de bloques de un sistema con dos procesadores. La memoria ocupa la posición central y puede comunicarse con cada procesador mediante DMA. La CPU es responsable del procesado necesario de datos para solucionar las tareas computacionales. El IOP proporciona un camino para la transferencia de datos entre varios periféricos y la memoria. La CPU tiene normalmente la tarea de iniciar el programa de E/S. Desde su puesta en marcha, el IOP opera independientemente de la CPU y continua con la transferencia de datos entre los dispositivos internos y la memoria. El formato de los datos de los dispositivos periféricos difiere frecuentemente de los de la memoria y la CPU. El IOP debe estructurar las palabras de los datos procedentes de las diferentes fuentes. Por ejemplo, puede ser necesario coger cuatro bytes de un dispositivo de entrada y empaquetarlos en una palabra de 32 bits antes de transferirlos a la memoria. Los datos se recogen en el IOP a una cierta velocidad y en una determinada cantidad mientras que la CPU ejecuta su propio programa. Después de ensamblarlos en una palabra de memoria, los datos se transfieren desde el IOP directamente a la memoria robando un ciclo de memoria a la CPU. De forma similar, una palabra de la memoria transferida al IOP se manda del IOP al dispositivo de salida a una cierta velocidad y cantidad.

La comunicación entre el IOP y los dispositivos conectados es similar al método de transferencia controlado por programa. La comunicación con la memoria es similar al método de DMA. La forma en que la CPU y el IOP se comunican con los otros depende del nivel de sofisticación del sistema. En sistemas de muy alta escala, cada procesador es independiente del resto, y cualquier procesador puede iniciar una operación. En la mayoría de los sistemas basados en procesador, la CPU es el procesador maestro y el IOP es el procesador esclavo. La CPU tiene asignada la tarea de inicializar todas las operaciones pero las instrucciones de E/S se ejecutan en el IOP. Las instrucciones de la CPU proporcionan las operaciones para empezar una transferencia de E/S y también comprobar las condiciones de status de E/S necesarias para tomar decisiones en las diversas actividades de E/S. La IOP, en cambio, típicamente realiza solicitudes a la CPU mediante interrupciones. También responde a las peticiones de la CPU colocando una palabra de status en una posición concreta de la memoria, para ser examinada posteriormente por la CPU. Cuando se desea realizar una operación de I/O, la CPU informa al IOP dónde encontrar el programa de E/S y luego deja los detalles de la transferencia al IOP.

A las instrucciones que lee el IOP de la memoria se suelen llamar *comandos*, para distinguirlos de las instrucciones que lee la CPU. Una instrucción y un comando tienen funciones

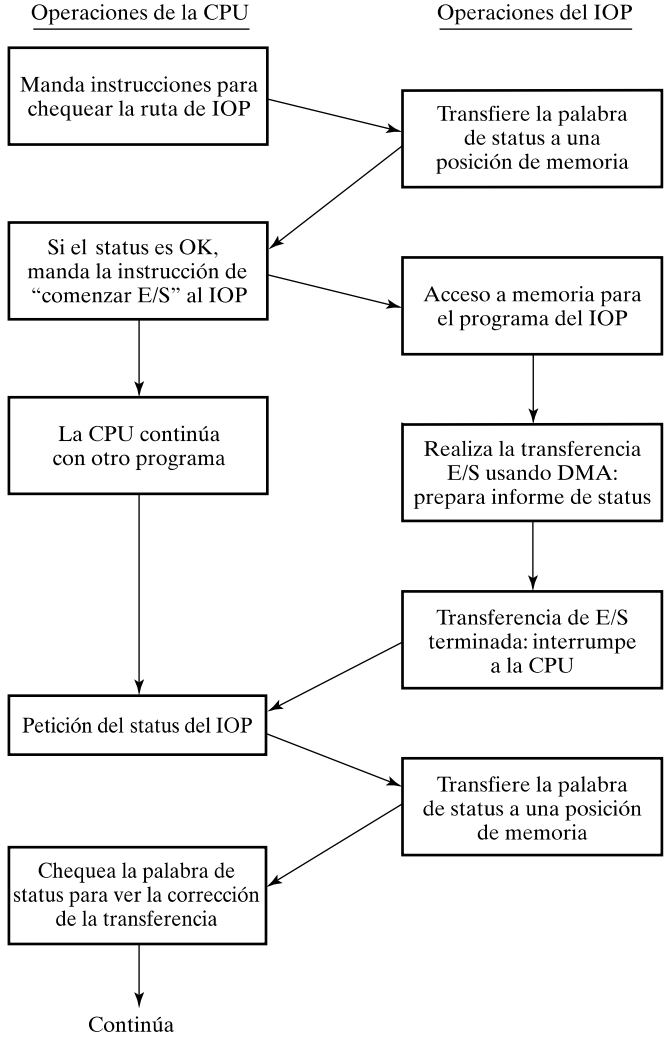


□ **FIGURA 13-21**
Diagrama de bloques de un sistema con procesadores de E/S

similares. Las palabras de los comandos constituyen el programa del IOP. La CPU informa al IOP donde puede encontrar los comandos en la memoria cuando es el momento de ejecutar el programa de E/S.

La comunicación entre la CPU y el IOP puede realizarse de diferentes formas, dependiendo del procesador en concreto que se utilice. En la mayoría de los casos, la memoria actúa como centro de mensajes, donde cada procesador deja información para los otros. Para apreciar la operación de un IOP típico, ilustraremos el método mediante el cual la CPU y el IOP se comunican uno con el otro. Este ejemplo simplificado omite algunos detalles para proporcionar una visión de los conceptos básicos.

La secuencia de operaciones se puede realizar según se muestra en el diagrama de flujo de la Figura 13-22. La CPU manda una instrucción para comprobar la ruta del IOP. El IOP responde insertando una palabra de status en la memoria de la CPU para su chequeo. Los bits de la



□ **FIGURA 13-22**
Comunicación CPU-IOP

palabra de status indican el estado del IOP y del dispositivo de E/S, como por ejemplo «estado de sobrecarga del IOP», «dispositivo ocupado con otra transferencia» o «dispositivo listo para transferencia de E/S». La CPU consulta la palabra de status de la memoria para decidir qué es lo siguiente. Si todo está en orden, la CPU manda la instrucción para comenzar la transferencia de E/S. La dirección de memoria recibida con esta instrucción le dice al IOP donde encontrar su programa.

Ahora la CPU puede continuar con otro programa mientras el IOP está ocupado con el programa de E/S. Ambos programas consultan la memoria mediante transferencias por DMA. Cuando el IOP termina la ejecución de su programa, manda una petición de interrupción a la CPU. La CPU responde enviando una instrucción para leer el status del IOP. Entonces, el IOP responde colocando el contenido de su informe de status en una posición concreta de la memoria. La palabra de status indica si la transferencia se ha terminado o si ha ocurrido algún error durante la transferencia. Por inspección de los bits de la palabra de status, la CPU determina si la operación de E/S se ha terminado satisfactoriamente, sin errores.

El IOP cuida de que todas las transferencias de datos entre las diversas unidades de E/S y la memoria mientras la CPU procesa otro programa. El IOP y la CPU compiten por el uso de la memoria, de forma que el número de dispositivos que pueden estar operando está limitado por el tiempo de acceso de la memoria. En la mayoría de los sistemas, no es posible que los dispositivos de E/S saturen la memoria pues la velocidad de la mayor parte de los dispositivos es mucho menor que la de la CPU. Sin embargo, varios dispositivos rápidos, como discos magnéticos o tarjetas gráficas, pueden utilizar un número apreciable de ciclos de memoria. En este caso, la velocidad de la CPU puede deteriorarse puesto que la CPU debe esperar con frecuencia a que el IOP proceda con las transferencias de memoria.

13-9 RESUMEN DEL CAPÍTULO

En este capítulo hemos presentado los dispositivos de E/S, llamados típicamente periféricos, y las estructuras digitales asociadas que les dan soporte, incluyendo buses de E/S, interfaces y controladores. Hemos estudiado la estructura de un teclado, un disco duro y una tarjeta gráfica. Hemos visto un ejemplo de una interfaz genérica de E/S y examinado la interfaz y el controlador de E/S de un teclado. Hemos presentado el USB como una solución alternativa para conectar varios dispositivos de E/S. Hemos considerado los problemas de tiempo entre sistemas con diferentes relojes y la transmisión de información vía serie y paralela.

También hemos visto los modos de transferir información y vimos cómo funcionan los modos más complejos, principalmente para relevar a la CPU de transferencias extensas, manejo de ejecución-robo. Las transferencias iniciadas mediante interrupción con varias interfaces de E/S llevan a un medio de establecer prioridades entre las fuentes de interrupción. La prioridad se puede manejar mediante software, lógica en una *daisy chain* serie o lógica de prioridad de interrupciones en paralelo. Los accesos directos a memoria logran hacer la transferencia de datos directamente entre la interfaz E/S y la memoria, con una pequeña participación de la CPU. Para terminar, los procesadores proporcionan una mayor independencia de la CPU del manejo de la E/S.

REFERENCIAS

1. PATTERSON, D. A., and J. L. HENNESSY: *Computer Organization and Design*: *The Hardware/Software Interface*. San Francisco, CA: Morgan Kaufmann, 1998.

2. VAN GILLUWE, F.: *The Undocumented PC*. Reading, MA: Addison-Wesley, 1994.

3. MESSMER, H. P.: *The Indispensable PC Hardware Book*. 2nd ed. Reading, MA: Addison-Wesley, 1995.

4. MindShare, Inc. (Don Anderson): *Universal Serial Bus System Architecture*. Reading, MA: Addison-Wesley Developers Press, 1997.

PROBLEMAS

El signo (+) indica problemas más avanzados y el asterisco indica que hay una solución disponible en la dirección de Internet: http://www.librosite.net/Mano.

13-1. *Encuentre la capacidad formateada de los discos duros descritos en la siguiente tabla:

Disco	Cabezas	Cilindros	Sectores/Pista	Bytes/Sector
A	1	1023	63	512
B	4	8191	63	512
C	16	16 383	63	512

13-2. Estime el tiempo necesario para transferir un bloque de 1 MB (2^{20} B) del disco a la memoria dada con los siguiente parámetros: tiempo de búsqueda, 8.5 ms; retardo de giro, 4.17 ms; tiempo del controlador, despreciable; tasa de transferencia, 100 MB/s.

13-3. Las direcciones asignadas a los cuatro registros de la interfaz de E/S de la Figura 13-5 son iguales a los equivalentes en binario de 240, 241, 242 y 243. Muestre el circuito externo que se debe conectar entre una dirección de 8 bits de E/S de la CPU y las entradas *CS*, *RS*0 y *RS*1 de la interfaz.

13-4. *¿Cuántas unidades de interfaz del tipo mostrado en la Figura 13.5 se pueden direccionar utilizando direcciones de 16 bits suponiendo

(a) que cada línea de selección de chip (*CS*) se conecta a una línea de dirección diferente?

(b) que los bits de direcciones están completamente decodificados para formar las entradas de selección del chip?

13-5. Se conectan seis unidades de interfaz, del tipo mostrado en la Figura 13-5, a la CPU que utiliza direcciones de E/S de ocho bits. Cada una de las seis entradas de selección de chip (*CS*) se conecta a una línea diferente de dirección. En concreto, la línea de dirección 0 se conecta a la entrada *CS* de la primera unidad de interfaz y la línea 5 se conecta a la entrada *CS* de la sexta unidad de interfaz. Las líneas de dirección 7 y 6 se conectan a las entradas *RS*1 y *RS*0 respectivamente, de todas las unidades de interfaz. Determine las direcciones de 8 bits de cada registro en cada interfaz (un total de 24 direcciones).

13-6. *Un tipo diferente de E/S no tiene entradas *RS*1 y *RS*0. Se pueden direccionar hasta dos registros utilizando señales separadas de E/S para lectura y escritura para cada dirección disponible. Suponga que el 50% de los registros de la interfaz con la CPU son sólo de lectura, el 25% de los registros son sólo de escritura y el 25% restante de los registros

son de lectura y escritura (bidireccionales) ¿Cuántos registros se pueden direccionar si la dirección contiene cuatro bits?

13-7. Una unidad de interfaz comercial utiliza nombres diferentes a los que aparecen en este texto para las líneas de *handshake* asociadas con la transferencia de datos desde el dispositivo de E/S a la unidad de interfaz. La línea de entrada de *handshake* a la interfaz se etiqueta como STB (*strobe*) la línea de salida de *handshake* de la interfaz se etiqueta como IBF (*input buffer full*, buffer de entrada lleno). Una señal activa a nivel bajo en STB carga el dato del bus de E/S al registro de datos de la interfaz. Un nivel alto en la señal IBF indica que la interfaz ha aceptado el dato. IBF pasa a nivel bajo después de una señal de lectura de E/S de la CPU cuando ésta lee el contenido del registro de datos.

(a) Dibuje un diagrama de bloques que muestre la CPU, la interfaz y el dispositivo de E/S con las interconexiones pertinentes entre las tres unidades.
(b) Dibuje un diagrama de tiempos para dicha transferencia.

13-8. *Suponga que las transferencias con *strobing* mostradas en la Figura 13-6 se realizan entre la CPU de la izquierda y una interfaz de E/S a la derecha. Hay una dirección procedente de la CPU para cada una de las transferencias, ambas se han iniciado por la CPU.

(a) Dibuje un diagrama de bloques que muestre las interconexiones para realizar las transferencias.
(b) Dibuje los diagramas de tiempos para las dos transferencias, suponga que la dirección se debe aplicar un tiempo antes de que el *strobe* se ponga a 1 y se quita un tiempo después de que el *strobe* se ponga a 0.

13-9. Suponga que las transferencias con *handshaking* mostradas en la Figura 13-7 se efectúan entre la CPU de la izquierda y la interfaz E/S de la derecha. Hay una dirección procedente de la CPU para cada una de las transferencias, ambas iniciadas por la CPU.

(a) Dibuje los diagramas de bloques que muestren las interconexiones para realizar las transferencias.
(b) Dibuje los diagramas de tiempos suponiendo que la dirección se debe aplicar un tiempo antes de que la petición se ponga a 1 y se quite un tiempo después de que la petición se ponga a 0.

13-10. *¿Cuántos caracteres por segundo se pueden transmitir en una línea de 57.600 baudios en cada uno de los siguientes modos? (Suponga caracteres de 8 bits).

(a) Transmisión asíncrona serie con dos bits de parada.
(b) Transmisión asíncrona serie con un bit de parada.
(c) Repita los apartados a y b para una línea de 115.200 baudios.

13-11. Esboce el diagrama de tiempos de 11 bits (similar a la Figura 13-8) que se transmite por una línea de comunicación asíncrona serie cuando se transmite el código ASCII de la letra E con paridad par. Suponga que el código ASCII del carácter se transmite empezando por el bit menos significativo, con el bit de paridad a continuación del código del carácter.

13-12. ¿Cuál es la diferencia entre la transferencia de información serie síncrona y asíncrona?

13-13. *Esboce la forma de ondas para el patrón de SYNC utilizado en USB y la correspondiente forma de onda NRZI. Explique por qué el patrón seleccionado es una buena elección para conseguir la sincronización.

13-14. La siguiente cadena de datos se transmite mediante USB:

01111111001000001111110111111101

(a) Suponiendo que no se utiliza el bit de relleno, esboce la forma de onda NRZI.
(b) Modifique la cadena aplicando el bit de relleno.
(c) Esboce la forma de onda NRZI para el resultado del apartado b.

13-15. *La palabra «Bye», codificada en ASCII de 8 bits, se transmite a una dirección del dispositivo (*device address*) 39 y una dirección de punto final (*endpoint address*) 2. Enumere los paquetes de salida y de Dato 0 y el paquete de *Handshake* para una parada antes de la codificación NRZI.

13-16. Repita el problema 13-15 para la palabra «Hlo» y un paquete de *Handshake* de tipo *No Acknowledge*.

13-17. ¿Cuál es la ventaja básica de utilizar una transferencia de datos iniciada por interrupción sobre una transferencia bajo control de programa sin interrupción?

13-18. *¿Qué sucede en la *daisy chain* con prioridad mostrada en la Figura 13-15 cuando el dispositivo 0 solicita una interrupción después de que el dispositivo 2 ha mandado su petición de interrupción a la CPU pero antes de que la CPU responda con un reconocimiento de interrupción?

13-19. Considere un procesador sin hardware de interrupción con prioridad. Cualquier fuente puede interrumpir al procesador, y cualquier petición de interrupción da lugar al almacenamiento de la dirección de retorno y a la bifurcación a una rutina de servicio a la interrupción común. Explique cómo se puede establecer una prioridad en el programa de servicio a las interrupciones.

13-20. *¿Qué cambios se necesitan hacer en la Figura 13-17 para tener cuatro valores de dirección de vector de interrupción (*VAD*) iguales a los binarios equivalentes de 024, 025, 026 y 027?

13-21. Repita el Problema 13-20 para los valores VAD de 224, 225, 226 y 227.

13-22. *Diseñe el hardware de interrupción con prioridad en paralelo para un sistema con seis fuentes de interrupción.

13-23. Una estructura con prioridad se diseña de forma que proporcione direcciones de vectores.

(a) Obtenga la tabla de verdad condensada de un codificador con prioridad de 16×4.
(b) Las cuatro salidas *w*, *x*, *y*, *z* del codificador con prioridad se utilizan para proporcionar una dirección de vector de 8 bits de la forma 10wxyz01. Enumere las 16 direcciones, empezando por la de mayor prioridad.

13-24. *¿Por qué las líneas de controlador de DMA son bidireccionales? ¿Bajo qué condiciones y para qué propósito se usan como salidas?

13-25. Es necesario transferir 1024 palabras de un disco magnético a una parte de la memoria que comienza en la dirección 2048. La transferencia se efectúa mediante DMA, según se muestra en la Figura 13-20.

(a) Indique los valores iniciales que debe transferir la CPU al controlador de DMA.

(b) Indique paso a paso la cantidad de acciones que se realizan durante la introducción de las dos primeras palabras.

CAPÍTULO

14

SISTEMAS DE MEMORIA

En el Capítulo 9 discutimos la tecnología RAM, incluyendo SRAM y DRAM, para la implementación de sistemas de memoria. En este capítulo profundizaremos más en lo que constituye el sistema de memoria de una computadora real. Comenzaremos partiendo de la premisa de que lo deseable es disponer de una memoria rápida y grande y después demostraremos cómo la implementación directa de esa memoria, para una computadora típica, es demasiado costosa y lenta. Como consecuencia, estudiaremos una solución más elegante en la que la mayoría de los accesos a memoria son rápidos (aunque alguno pueda ser lento) y la memoria aparenta ser grande. Esta solución se basa en dos conceptos: memoria caché[1] y memoria virtual. Una memoria caché es una memoria pequeña y rápida, dotada de mecanismos hardware de control especiales que permiten que pueda hacerse cargo de la mayoría de los accesos a memoria ordenados por la CPU, con un tiempo de acceso del orden de un periodo de reloj de la CPU. La memoria virtual, implementada mediante hardware y software, emplea una memoria principal intermedia de tamaño medio (usualmente DRAM) y se comporta como una memoria principal de tamaño mayor, pero conserva los tiempos de acceso de la memoria intermedia para la gran mayoría de los accesos. El medio de almacenamiento real para la mayor parte del código y datos en la memoria virtual es un disco duro. Dado que existe una progresión de componentes en el sistema de memoria, siendo estos componentes de capacidad creciente, pero de tiempos de acceso cada vez menores (caché, memoria principal y disco duro), se dice que existe una jerarquía de memoria.

En el diagrama de una computadora genérica al principio del Capítulo 1 existe una cantidad de componentes relacionados estrechamente con la jerarquía de memoria. Dentro del procesador está la unidad de manejo de memoria (MMU, Memory Management Unit), que es hardware dedicado a soportar la memoria virtual. También en el procesador dispone de su propia caché interna. Puesto que esta caché es demasiado pequeña para desarrollar plenamente la función de la caché existe una memoria caché externa conectada al bus de la CPU. Por supuesto, la RAM y, debido a la presencia de memoria virtual, el disco duro, el interfaz del bus y el controlador de disco forman parte del sistema de memoria.

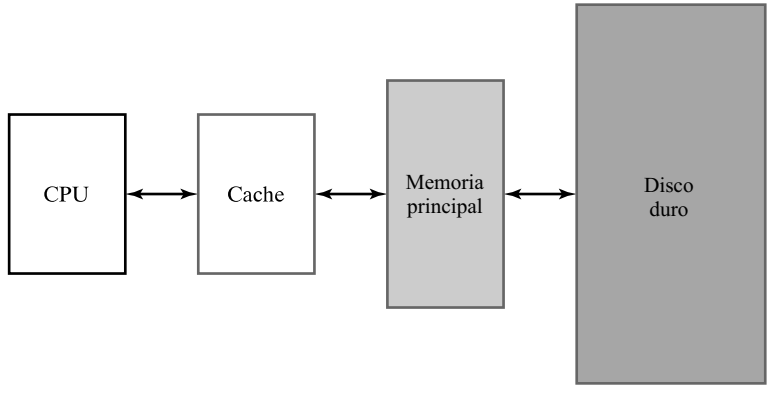
[1] *N. del T.:* El término inglés «caché» no suele ser traducido al español.

14-1 JERARQUÍA DE MEMORIA

En la Figura 14-1 se muestra el diagrama de bloques genérico de una jerarquía de memoria. El nivel más bajo de la jerarquía es una memoria pequeña, pero rápida, denominada caché. Para que el funcionamiento de la jerarquía de memoria completa sea adecuado es necesario que una fracción grande de los accesos a instrucciones y datos realizados por la CPU sean atendidos por la caché. En el siguiente nivel superior de esta jerarquía se encuentra la memoria principal. La memoria principal sirve directamente una parte de los accesos de la CPU no satisfechos por la caché. Además la caché busca todos sus datos, algunos de los cuales pasan a la CPU, de la memoria principal. En el nivel superior de la jerarquía se encuentra el disco duro, que únicamente es accedido en el caso poco frecuente de que el dato o instrucción buscado no se encuentre en la memoria principal.

En esta jerarquía de memoria, dado que la CPU encuentra la mayor parte de las instrucciones y los operandos en la caché «ve» una memoria rápida la mayor parte del tiempo. Ocasionalmente, cuando una palabra debe ser leída de la memoria principal, la operación de búsqueda lleva un tiempo algo mayor. Muy raramente alguna palabra debe ser buscada en el disco duro, búsqueda que consume un tiempo muy largo. En este último caso la CPU, mediante una interrupción, pase el control a un programa capaz de leer un bloque de palabras desde el disco duro. En media la situación suele ser satisfactoria, arrojando un tiempo de acceso promedio similar al de la caché. Además la CPU ve un espacio de direcciones de memoria considerablemente mayor que el de la memoria principal.

Manteniendo en mente esta noción de la jerarquía de memoria vamos a considerar un ejemplo que ilustra el potencial de esta jerarquía. De todos modos debemos clarificar primero un asunto: en la mayoría de los juegos de instrucciones el objeto menor que se puede direccionar es un byte, no una palabra. Para una determinada operación de lectura o escritura se suele especificar si el dato a acceder es un byte o una palabra mediante el código de operación (*opcode*). El direccionamiento de bytes trae a colación algunos aspectos y detalles del hardware importantes, pero que si se expusiesen en este punto del texto lo complicarían de forma innecesaria. De este modo, y por simplicidad, hemos considerado hasta ahora que las direcciones de memoria almacenan una palabra. Por contra, tal y como es práctica habitual, en este capítulo supondremos que los datos almacenados en una dirección son bytes. Aun así y para evitar liosas explicaciones sobre el manejo de los bytes, seguiremos suponiendo que los datos se mueven alrededor de la CPU en forma de palabras o grupos de palabras. Esta forma de proceder oculta algunos detalles



□ **FIGURA 14-1**
Jerarquía de memoria

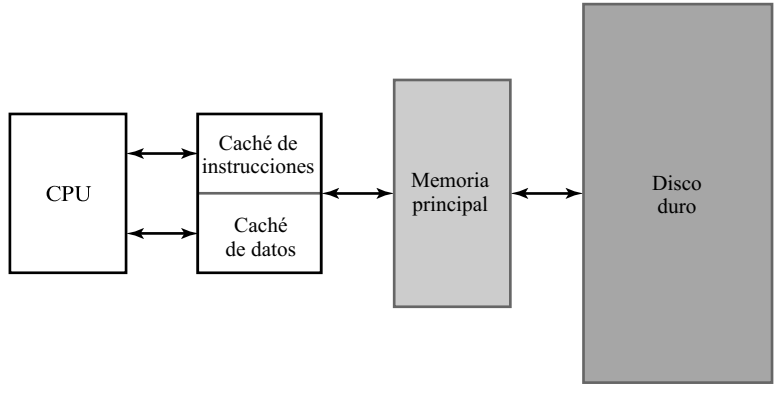
del hardware que pueden distraernos del objetivo principal del discurso, pero que en cualquier caso deben ser conocidos por el diseñador del hardware. Para llevar a cabo esta simplificación, si existen bytes en una palabra ignoraremos los *b* bits de menor pero de la dirección. Dado que estos bits no se necesitan para acceder a la dirección mostraremos su valor siempre a cero. Para los ejemplos que aparecerán *b* siempre será 2, con lo que siempre aparacerán dos ceros.

En la Sección 12-3 la CPU segmentada[2] trabaja con direcciones de memoria de 32 bits y es capaz, en caso necesario, de acceder a una instrucción y a un dato en sendos ciclos de 1 ns de duración del reloj. Además supusimos que las instrucciones y los datos se buscaban en dos memorias distintas. Para dar soporte a estas suposiciones en este capítulo partiremos de que la memoria está dividida en dos partes, una mitad para instrucciones y la otra mitad para datos. Cada una de estas mitades debe presentar un tiempo de acceso de 1 ns. Además, si empleamos todos los 32 bits de las direcciones, la memoria debe contener bytes (4 GB) de información. Así el objetivo es disponer de dos memorias de 2 GB, cada una con un tiempo de acceso de 1 ns.

¿Es esta memoria realista en términos de la presente (2003) tecnología de computadoras? La memoria habitual se construye a partir de módulos de DRA; de tamaño comprendido entre 16 y 64 MB. El tiempo de acceso típico es de alrededor de 10 ns. De este modo, nuestras dos memorias de 2 GB tendrían un tiempo de acceso algo mayor que 10 ns. Este tipo de memoria es demasiado cara y trabaja a la décima parte de la velocidad deseada. Por ello nuestro objetivo debe ser alcanzado por otro camino, lo que nos llevará a explorar la jerarquía de memoria.

Comenzaremos suponiendo una jerarquía con dos cachés, una para instrucciones y otra para datos, como se ve en la Figura 14-2. El uso de dos cachés permitirá, si estas cachés son suficientemente rápidas, que una instrucción y un operando puedan ser leídos, o una instrucción leída y un resultado almacenado en un único ciclo de reloj. En términos de la computadora genérica, supondremos que estas cachés son internas, por lo que podrán trabajar a velocidades comparables a la de la CPU. Así, la búsqueda en la caché de instrucciones y la búsqueda o el almacenamiento en la caché de datos pueden realizarse en 2 ns. De este modo la mayor parte de las búsquedas y almacenamientos se realizan sobre estas cachés y necesitarán 2 ciclos de reloj de la CPU para llevarse a cabo. Supongamos además que es suficiente con que el 95 % de los accesos se realicen empleando esos 2 ns y que el 5 % restante de los accesos a memoria necesitan 10 ns para efectuarse. Entonces el tiempo de acceso medio es:

$$0.95 \times 2 + 0.05 \times 10 = 2.4 \text{ ns}$$



□ **FIGURA 14-2**
Ejemplo de jerarquía de memoria

[2] *N. del T.:* Se ha empleado el término «segmentado» para designar la expresión inglesa «pipelined».

Esto significa que, en 19 de cada 20 accesos a memoria, la CPU trabaja a su máxima velocidad, mientras que la CPU debe esperar durante 10 ciclos de reloj en uno de cada 20 accesos a memoria. Esta espera puede realizarse deteniendo la segmentación de la CPU. De esta forma hemos satisfecho nuestro objetivo de que «la mayoría» de los accesos a memoria necesiten 2 ns. Pero aún tenemos el problema del coste de una memoria grande.

Supóngase ahora que, además de la poco frecuente espera para un dato de la memoria principal que necesitará mas de 10 ns, también debemos aceptar el caso extremadamente poco frecuente de un acceso al disco duro que necesite 13 ms = 1.3×10^7 ns. Supongamos además que disponemos de información que nos indica que alrededor del 95% de los accesos se hacen sobre la caché y que el 4.999995% se realizan sobre la memoria principal. Con esta información podemos estimar el tiempo de acceso medio como:

$$0.95 \times 2 + 0.04999995 \times 10 + 5 \times 10^{-8} \times 1.3 \times 10^7 = 3.05 \text{ ns}$$

De este modo un acceso requiere en promedio 3 ciclos del reloj de 1 ns de la CPU, pero este es aproximadamente un tercio del tiempo de acceso de la memoria principal y, de nuevo, 19 de cada 20 accesos a memoria se realizan en 2 ns. Así hemos alcanzado un tiempo de acceso medio de 3.05 ns para una estructura de memoria con una capacidad de 2^{32} bytes, no lejos del objetivo de partida. Aun más, el coste de esta jerarquía de memoria es decenas de veces menor que el del enfoque basado en una memoria grande y rápida.

Así parece que el objetivo original de disponer de una memoria rápida y grande se ha aproximado mediante una jerarquía de memoria a un precio razonable. Pero durante este desarrollo se han hecho algunas suposiciones, que en el 95% de los casos la palabra buscada está en lo que denominamos caché y que el 99.999995% del tiempo las palabras están en la caché o en la memoria principal, estando en los casos restantes en el disco duro. En lo que queda de capítulo estudiaremos qué presunciones como ésta son usualmente válidas y exploraremos el hardware y software asociado necesarios para alcanzar los objetivos de la jerarquía de memoria.

14-2 LOCALIDAD DE REFERENCIA

En la sección anterior indicamos que el éxito de la jerarquía de memoria se basaba en suposiciones que eran críticas para alcanzar la apariencia de una memoria grande y rápida. Ahora nos ocuparemos de los fundamentos que nos permiten realizar esas suposiciones y que se denomina localidad de referencia. Aquí «referencia» significa referencia a la memoria para acceder a las instrucciones o para leer o escribir operandos. El término «localidad» se refiere a los instantes relativos en los que se accede a las instrucciones o los datos (localidad temporal) y las posiciones relativas en las que éstos residen en memoria (localidad espacial).

Primeramente consideraremos la naturaleza típica de un programa. Un programa frecuentemente contiene muchos bucles. En un bucle una secuencia de instrucciones es ejecutada muchas veces antes de que el programa salga del bucle y se mueva a otro bucle o a código línea a línea sin bucles. Aún más, los bucles están muchas veces anidados en una jerarquía en la que existen bucles que contienen bucles y así sucesivamente. Supongamos que tenemos un bucle de 8 instrucciones que debe ser ejecutado 100 veces. Entonces durante 800 ejecuciones todas las búsquedas de instrucción ocurrirán en solamente ocho direcciones de memoria. Así cada una de estas 8 direcciones es visitada 100 veces durante el tiempo en el que el bucle es ejecutado. Este es un ejemplo de localidad temporal, en el sentido de que si una dirección es accedida es probable que sea accedida más veces en un futuro próximo. De igual forma, es también razonable que las direcciones de instrucciones sucesivas estén en orden secuencial. Así si una dirección es

accedida por una instrucción, las direcciones próximas serán posiblemente direccionadas durante la ejecución del bucle. Este es un ejemplo de localidad espacial.

Para el caso de acceso a operandos también tienen sentido la localidad espacial y temporal. Por ejemplo en un cálculo sobre un arreglo de números existirán múltiples accesos a las posiciones de muchos de los operandos, lo que da lugar a localidad temporal. Además, según se desarrolle el calculó, cuando una dirección de un determinado número sea accedida es probable que las direcciones secuenciales a ella sean también accedidas para otros números del arreglo, lo que da lugar a localidad espacial.

De la discusión anterior podemos conjeturar que en los programas de computadora existe una significativa localidad de las referencias. Para verificar esto definitivamente es necesario estudiar patrones de ejecución de programas reales. Estos estudios han demostrado la presencia de una localidad espacial y temporal muy significativa y juegan un papel importante en el diseño de cachés y sistemas de memoria virtual.

La siguiente cuestión a responder es: ¿cuál es la relación entre localidad de referencia y jerarquía de memoria? Para examinar esta cuestión consideraremos de nuevo la búsqueda de una instrucción dentro de un bucle y nos centraremos en la relación entre la caché y la memoria principal. Inicialmente supondremos que las instrucciones están presentes únicamente en la memoria principal y que la caché está vacía. Cuando la CPU busca la primera instrucción de un bucle, la obtiene de la memoria principal. Pero la instrucción y una porción de su dirección, denominada etiqueta de dirección, son almacenadas en la caché. ¿Qué ocurre para las siguientes 99 ejecuciones de esta instrucción? La respuesta es que cada instrucción puede ser buscada en la caché, lo que da lugar a unos tiempos de acceso mucho más rápidos. Aquí está trabajando la localidad temporal. La instrucción que fue buscada una vez tenderá a ser usada de nuevo, pero ahora está presente en la caché y su acceso será más rápido.

Adicionalmente, cuando la CPU busca una instrucción de la memoria principal, la caché almacena instrucciones próximas en su SRAM. Ahora supongamos que estas instrucciones próximas incluyen el bucle entero de 8 instrucciones de este ejemplo. Entonces todas las instrucciones están en la caché. Trayendo este bloque de instrucciones la caché puede ahora explotar la localidad espacial. La caché aprovecha el hecho de que la ejecución de la primera instrucción implica la ejecución de instrucciones de direcciones próximas haciendo que las instrucciones próximas estén disponibles para su rápido acceso.

En nuestro ejemplo, cada instrucción es buscada en la memoria principal exactamente una vez para las 100 ejecuciones del bucle. Todas las demás búsquedas tienen lugar sobre la cache. Así, en este ejemplo concreto, al menos el 99% de las instrucciones a ejecutar son buscadas en la caché, de modo que la velocidad de ejecución de estas instrucciones está gobernada casi exclusivamente por el tiempo de acceso de la caché y por la velocidad de la CPU, y en mucha menor medida por el tiempo de acceso de la memoria principal. Sin localidad temporal ocurrirían muchos más accesos a la memoria principal, ralentizando el sistema.

Una relación similar a la que existe entre caché y memoria principal existe entre memoria principal y disco duro. De nuevo son de interés tanto la localidad espacial como la temporal, aunque en este caso a una escala mucho mayor. Los programas y los datos son buscados en el disco duro y los datos son escritos en el disco duro en bloques que pueden ser de k-palabras o M-palabras. Idealmente, una vez que el código y los datos de partida de un programa residen en la memoria principal, no es necesario que el disco vuelva a ser accedido hasta que sea necesario almacenar los resultados finales del programa. Pero esto solo será cierto si todo el código y datos del programa, incluyendo los datos intermedios, residen completamente en la memoria principal. Sino, entonces será necesario traer código desde el disco duro y leer y escribir datos en el disco durante la ejecución del programa. Las palabras son escritas y leídas en el disco duro

en bloques denominados páginas. Si el movimiento de páginas entre la memoria principal y el disco es transparente al programador, entonces parecerá que la memoria principal es suficientemente grande como para almacenar el programa completo y todos sus datos. Por ello este funcionamiento automático es denominado memoria virtual. Durante la ejecución de un programa, si una instrucción no se encuentra en la memoria principal, el flujo del programa se desvía para traer a la memoria principal la página que contiene la instrucción. Entonces la instrucción puede moverse a la memoria principal y ejecutada. Los detalles de esta operación y el hardware y software involucrados serán descritos en la Sección 14-4.

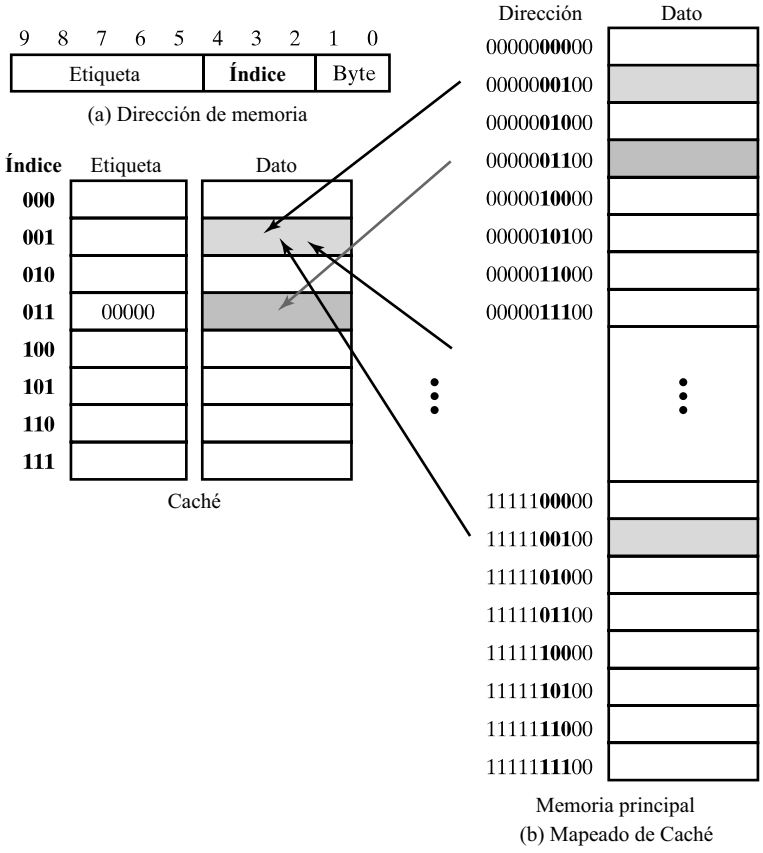
En suma, la localidad de las referencias es clave para el éxito de los conceptos de memoria caché y memoria virtual. En la mayoría de los programas, la localidad de las referencias está presente en gran medida. Pero ocasionalmente uno puede encontrar un programa que, por ejemplo, debe acceder frecuentemente a una cantidad de datos que no puede ser almacenada en memoria. En este caso la computadora consume la mayor parte del tiempo moviendo información entre la memoria principal y el disco duro, sin prácticamente realizar ningún cálculo. Un síntoma de este problema es un sonido continuo, proveniente del disco duro y producido por el movimiento constante entre pistas de las cabezas, denominado *thrashing*[3].

14-3 MEMORIA CACHÉ

Para ilustrar el concepto de memoria caché supondremos el caso de una caché muy pequeña de 8 palabras de 32 bits y una memoria principal pequeña de 1 KB (256 palabras), como se muestra en la Figura 14-3. Ambas son demasiado pequeñas para ilustrar una situación real, pero sus tamaños permitirán ilustrar los conceptos más fácilmente. Las direcciones en la caché son de 3 bits. las de memoria 10. De las 256 palabras de la memoria principal solo 8 al mismo tiempo pueden almacenarse en la caché. Para que la CPU pueda direccionar una palabra de la caché, es necesario que ésta almacene alguna información que permita identificar la dirección de la palabra en la memoria principal. Si considaremos el ejemplo del bucle en la sección anterior, parece claro que lo deseable es mantener el bucle completo en la caché, de forma que todas las instrucciones puedan ser buscadas en la caché mientras el programa ejecuta las diversas pasadas del bucle. Las instrucciones en el bucle ocupan direcciones de palabra consecutivas. Por ello es deseable que la caché almacene palabras de direcciones consecutivas de la memoria principal. Una forma sencilla de alcanzar este objetivo es hacer que los bits 2 a 4 de las direcciones de la memoria principal sean la dirección en la caché. Nos referiremos a estos bits como el índice (*index*), como se ve en la Figura 14-3. Note que el dato de la dirección 0000001100 de la memoria principal debe almacenarse en la dirección de la caché 011. Los 5 bits más significativos de la dirección de la memoria principal, denominados la etiqueta (*tag*), se almacenan en la caché junto con el dato. Continuando con el ejemplo, encontramos que para la dirección de la memoria principal 0000001100 la etiqueta es 00000. La etiqueta y el índice (o dirección en la caché) y el 00 correspondiente al campo de byte identifican una dirección de la memoria principal.

Supongamos que la CPU busca la instrucción en la dirección 000001100 de la memoria principal. Esta instrucción podrá provenir de la caché o de la memoria principal. La caché separa la etiqueta 0000 de la dirección en la caché 011, internamente busca la etiqueta y la palabra almacenada en la posición 011 de la caché y compara la etiqueta almacenada con la porción de la dirección generada por la CPU correspondiente a la etiqueta. Si la etiqueta encontrada es 00000

[3] *N. del T.:* El término inglés *thrashing* a veces se traduce, algo informalmente, por «rascado».



□ **FIGURA 14-3**
Caché mapeado directo

ocurre un acierto (*match*) y la palabra almacenada, leída de la memoria cache, es la instrucción deseada. Así, el control de la caché coloca esta palabra en el bus de la CPU, completando la operación de búsqueda. Este caso en el que la palabra buscada en la memoria se encuentra en la caché se denomina *cache hit*. Si la etiqueta almacenada en la memoria caché no es 00000, existe un desacuerdo de la etiqueta y el control de la memoria caché indica a la memoria principal que debe aportar la palabra buscada, que no está alojada en la cache. Esta situación se denomina *cache miss*[4]. Para que una caché sea efectiva es necesario evitar al máximo los accesos, lentos, a la memoria principal, de modo que haya muchos más *cache hits* que *cache miss*.

Cuando ocurre un *cache miss* en una operación de búsqueda, la palabra leída de la memoria principal no sólo se coloca en el bus de la CPU. La caché también lee esta palabra y su etiqueta y la almacena para futuros accesos. En nuestro ejemplo, la etiqueta 00000 y la palabra de la memoria se escribirán en la posición 011, en previsión de futuros accesos a la misma dirección de memoria. El manejo de las escrituras en memoria será tratado más adelante en este capítulo.

[4] *N. del T.:* Las expresiones inglesas «cache hit» y «cache miss» no suelen traducirse al español, aunque a veces se emplean las expresiones «acierto de caché» y «fallo de caché» para designarlas.

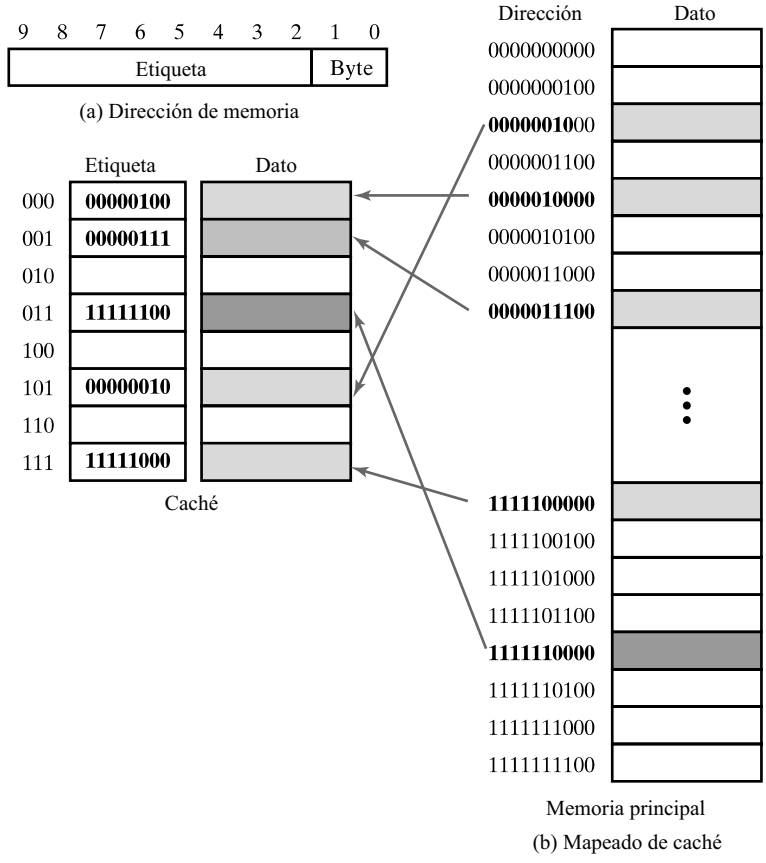
Mapeado de la caché

El ejemplo recién descrito emplea una asociación determinada entre las direcciones de la memoria principal y las de la caché, concretamente que los 3 últimos bits de la dirección de palabra en la memoria principal son la dirección en la memoria caché. Además sólo hay una posición en la caché para las 25 posiciones de memoria principal que tienen sus 3 últimos bits en común. Este mapeado, mostrado en la Figura 14-3 en el que sólo una posición determinada de la caché puede contener la palabra de una dirección particular de la memoria principal se denomina mapeado directo.

Este mapeado directo no siempre produce los mejores resultados. En el ejemplo de búsqueda de una instrucción de un bucle supóngase que instrucciones y datos se almacenan en la misma y que el dato de la posición 1111101100 se usa frecuentemente. Entonces, cuando se busca la instrucción 0000001100 lo normal es que la posición 011 de la caché contenga el dato de 1111101100, con etiqueta 11111. Ocurre un *cache miss* y entonces la etiqueta 11111 es reemplazada por 00000 y el dato reemplazado por la instrucción. Pero la próxima vez que se necesite el dato volverá a ocurrir un *cache miss*, dado que la posición de la caché está ahora ocupada por la instrucción. Durante la ejecución del bucle las búsquedas del dato y de la instrucción ocasionarán gran cantidad de *cache misses*, lo que disminuirá fuertemente la velocidad de procesado. Para resolver este problema exploraremos mapeados alternativos de la caché.

En el mapeado directo 25 direcciones de la memoria principal se mapean a una única dirección de la caché con la que comparten los 3 bits menos significativos. Estas posiciones están destacadas en gris en la Figura 14-3 para el índice 001. Tal como se ve, en un determinado instante del tiempo, sólo una de las 25 direcciones puede tener la palabra almacenada en la dirección 001 de la caché. Por el contrario, supóngase que es posible que las posiciones de la memoria principal puedan ser mapeadas en posiciones arbitrarias de la caché. Entonces cualquier posición de la memoria principal se puede mapear a cualquiera de las 8 posiciones de la caché. Esto significa que la etiqueta será ahora la dirección completa de la palabra en la memoria principal. Examinaremos ahora el funcionamiento de esta caché de la Figura 14-4 que ahora presenta un mapeado completamente asociativo. Note que ahora hay 2 direcciones de la memoria principal, 0000010000 y 111111000, con los bits 2 a 4 iguales a 100, entre las etiquetas de la caché. Estas 2 direcciones no pueden estar presentes simultáneamente en una caché con mapeado directo, ya que las 2 ocuparían la dirección de caché 100. De esta forma, ahora se evita la sucesión de *cache misses* producida por la búsqueda alternativa del dato y de la instrucción con igual índice, puesto que ahora los 2 pueden estar almacenados en la caché.

Supóngase ahora que la CPU debe buscar una instrucción de la posición 0000010000 de la memoria principal. Esta instrucción puede ser entregada por la caché o por la memoria principal. Puesto que puede estar en la caché ésta debe comparar 00000100 con cada una de las 8 etiquetas que almacena. Una forma de hacer esto es leer sucesivamente las etiquetas y palabras asociadas de la memoria caché y comparar las etiquetas con 00000100, Si alguna comparación es satisfactoria, como ocurrirá para la dirección dada en la posición 000 de la Figura 14-4, entonces se produce un *cache hit*. El control de la caché colocará entonces la palabra en el bus de la CPU, completando la operación de búsqueda. Si la etiqueta almacenada en la caché no es 00000100 entonces hay un desacuerdo de las etiquetas y el control de la caché busca las siguientes etiqueta y palabra almacenada. En el peor de los casos, un acuerdo de etiquetas en la posición 111, requiere 8 búsquedas en la caché hasta que se produzca el *cache hit*. A razón de 2 ns cada búsqueda esto requiere al menos 16 ns, aproximadamente la mitad del tiempo necesario para obtener la instrucción desde la memoria principal. De este modo, la lectura sucesiva de etiquetas y palabras de la caché hasta que se produzca un acierto no parece un enfoque demasiado

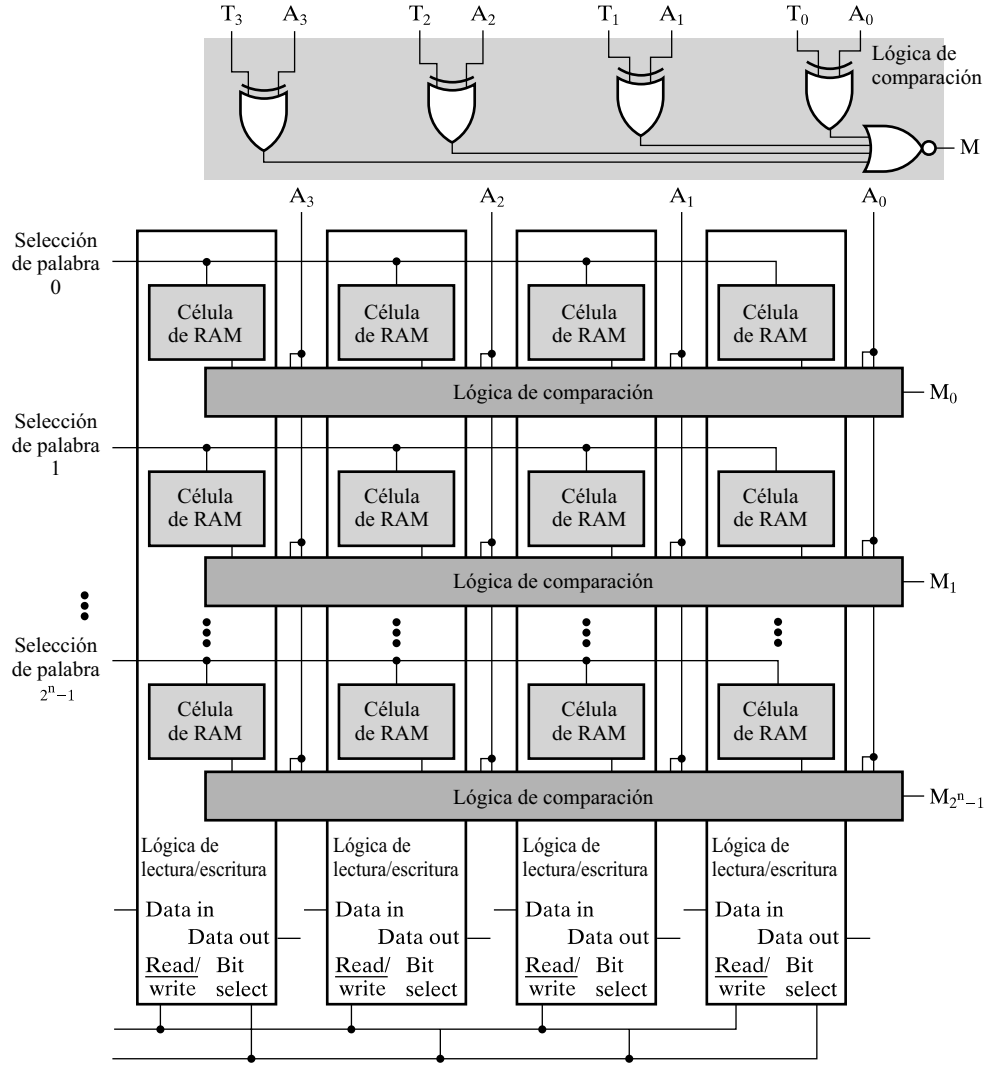


□ **FIGURA 14-4**
Caché completamente asociativo

interesante. En su lugar se emplea una estructura denominada memoria asociativa que implementa la parte de comparación de etiquetas de la memoria caché.

La Figura 14-5 muestra una memoria asociativa para una caché con etiquetas de 4 bits. El mecanismo para escribir etiquetas en la memoria es el habitual. De la misma forma, las etiquetas pueden ser leídas mediante una operación de lectura convencional. De este modo la memoria asociativa puede usar el *slice* RAM presentado en el Capítulo 9. Se añade a cada fila de almacenamiento de etiquetas una lógica de comparación. La implementación de esta lógica y su conexión a las células de la RAM se muestra en la figura. La lógica de comparación realiza una comparación de igualdad entre la etiqueta *E* y la dirección *D* aplicada por la CPU. La lógica de comparación está formada por una puerta XOR por cada bit y una puerta NOR que combina las salidas de todas las XOR. Si todos los bits de la etiqueta y de la dirección son iguales, todas las salidas de las XOR valen 0 y la salida de la NOR 1, indicando que hay una coincidencia. Si existe alguna diferencia en cualquiera de los bits de la etiqueta y de la dirección, entonces al menos una de las XOR entregará un 1 y la NOR entregará un 0, indicando el desacuerdo.

Puesto que todas las etiquetas son distintas sólo existen dos situaciones posibles en la memoria asociativa: o existe un acierto y hay un 1 en la salida de comparación de una etiqueta y 0 en las demás o todas las salidas de comparación son 0. En una memoria asociativa que almacene las etiquetas de la caché las salidas de la lógica de comparación atacan las líneas de palabra



□ **FIGURA 14-5**
Memoria asociativa para etiquetas de 4 bit

de la memoria de palabras a leer. Una señal debe indicar si ha ocurrido un *cache hit* o un *cache miss*. Si esta señal vale 1 para un *cache hit* y 0 para un *cache miss* puede ser generada como una OR de las salidas de comparación. En caso de un *cache hit* un 1 en Hit/$\overline{\text{miss}}$ coloca la palabra en el bus de memoria de la CPU, mientras que en caso de *cache miss* un 0 en Hit/$\overline{\text{miss}}$ indica a la memoria principal que debe entregar la palabra direccionada.

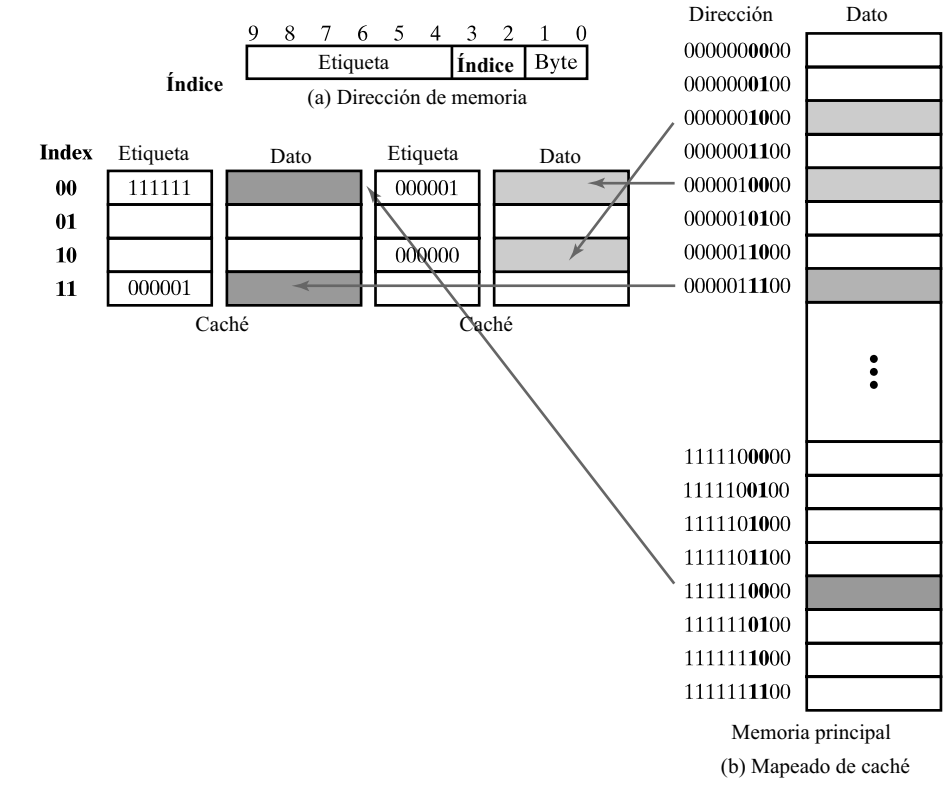
Como en el caso de la caché con mapeado directo discutido antes, la caché completamente asociativa debe capturar la palabra y su etiqueta y almacenarlas para futuros accesos. Pero ahora surge un nuevo problema: ¿en qué posición de la caché se almacenan la etiqueta y el dato? Además de seleccionar el tipo de mapeado el diseñador de la caché debe seleccionar la estrategia a emplear para determinar la dirección en la que se almacenarán nuevas etiquetas y datos. Una posibilidad es elegir la posición de forma aleatoria. La dirección de 3 bits puede ser leída

de una estructura hardware sencilla que genere un número que satisfaga algunas propiedades de los números aleatorios. Una estrategia algo más elaborada sería: «primero en entrar, primero en salir» (*First In, First Out, FIFO*). En este caso la posición seleccionada para ser reemplazada sería la que lleve ocupada más tiempo, basándonos en la noción de que la entrada más antigua es la que, con mayor probabilidad, haya dejado de ser usada. Otro enfoque para resolver este problema de forma aun más directa es el denominado «usado menos recientemente» (*Least Recently Used, LRU*). En este enfoque la entrada que se reemplaza es la que lleve sin usarse más tiempo. La razón de ser de este enfoque es que la entrada de la caché que lleve más tiempo si usarse es la que menos probabilidades tiene de ser usada en el futuro. Aunque el enfoque LRU arroja mejores prestaciones para las cachés, la diferencia entre este y otros enfoques no es demasiado grande, pero su implementación es costosa. Por ello el enfoque LRU es a menudo sólo aproximado, si es que llega a usarse en algún caso.

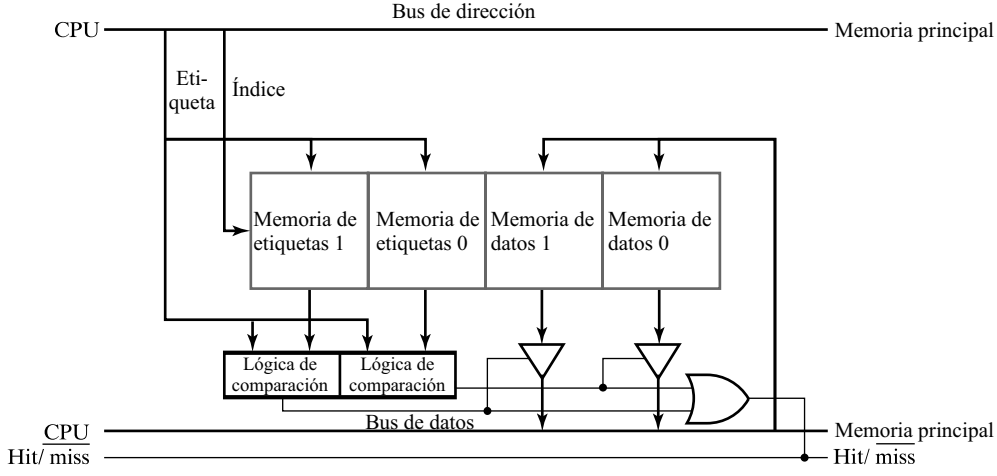
Existen además algunos aspectos de coste y prestaciones a considerar en la caché completamente asociativo. Aunque ésta presenta la mayor flexibilidad y mejores prestaciones, no está claro que su coste esté justificado. De hecho, existe un mapeado alternativo, que es un compromiso entre al mapeado directo y el completamente asociativo, que presenta mejores prestaciones y elimina el coste de la mayoría de la lógica de comparación. En este mapeado los bits bajos de la dirección funcionan como en el mapeado directo; pero para cada combinación de estos bits bajos, en vez de existir una única posición en la caché existe un conjunto de posiciones. Como ocurre en el mapeado directo las etiquetas y palabras se leen de las posiciones de la memoria de la caché direccionadas por los bits de menor peso de la dirección. Por ejemplo, si el tamaño del conjunto es 2, entonces se leen dos etiquetas y dos palabras simultáneamente. Estas etiquetas se comparan simultáneamente con la dirección entregada por la CPU mediante sólo dos circuitos de comparación. Si una de las etiquetas se compara satisfactoriamente con la dirección entonces la palabra asociada se entrega a la CPU a través del bus de memoria. Si ninguna de las etiquetas casa con la dirección entonces se envía una señal de *cache miss* a la CPU y a la memoria principal. Dado que existen conjuntos de posiciones y la asociatividad se aplica sobre estos conjuntos, esta técnica se denomina mapeado asociativo por conjuntos (*set-associative mapping*). Este mapeado con tamaño de conjuntos *s* se denomina mapeado asociativo por conjuntos de *s* vías (*s-way set-associative mapping*).

La Figura 14-6 muestra una caché asociativa por conjuntos de 2 vías. Existen 8 posiciones en la caché distribuidas en 4 filas de dos posiciones cada una. Las filas se direccionan mediante un índice de 2 bits y contienen etiquetas formadas por los restantes 6 bits de la dirección de la memoria principal. La entrada en la caché para una dirección de la memoria principal debe localizarse en una fila concreta de la caché, pero puede estar en cualquiera de las dos columnas. El la figura las direcciones son las mismas que en la caché completamente asociativa de la Figura 14-4. Note que no se muestra ningún mapeado para la dirección de memoria principal 1111100000, dado que las dos células de la caché en el conjunto 00 están ocupadas por las direcciones 00000100000 y 1111110000. Para poder albergar la dirección 1111100000 el tamaño de los conjuntos debiera ser al menos 3. Este ejemplo ilustra el caso en el que la menor flexibilidad de la caché asociativa por conjuntos, comparada con la caché completamente asociativa, tiene impacto sobre las prestaciones del circuito. Este impacto es menor según se aumenta el tamaño de los conjuntos.

La Figura 14-7 es una sección del diagrama de bloques de la caché asociativa por conjuntos de la Figura 14-6. El índice se usa para direccionar cada fila de la memoria de la caché. Las dos etiquetas leídas de la memoria de etiquetas son comparadas con la parte de la etiqueta de la dirección en el bus de la CPU. Si ocurre un acierto el buffer de tres estados de la memoria de datos correspondiente se activa, colocando los datos en el bus de la CPU. Además, la señal



□ **FIGURA 14-6**
Caché asociativa por conjuntos de 2 vías



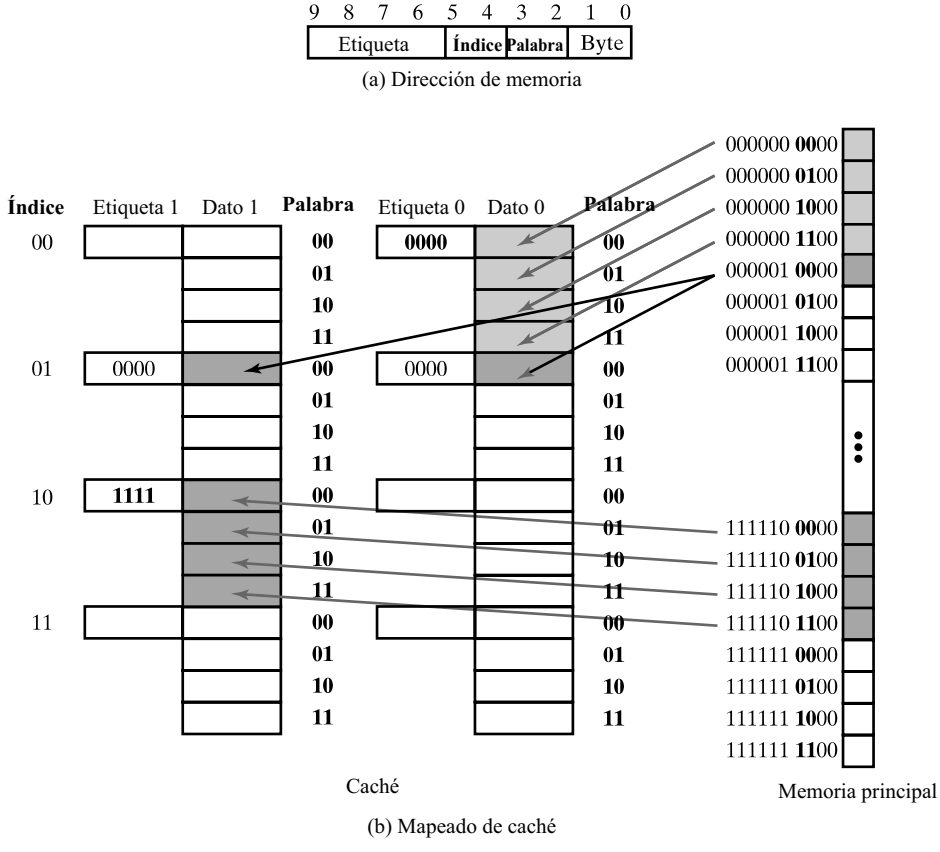
□ **FIGURA 14-7**
Diagrama de bloques parcial del hardware de una caché asociativa por conjuntos

de acierto origina que la salida de la puerta OR *Hit*/$\overline{miss}$ se ponga a 1, indicando un *cache hit*. Si no ocurre el acierto entonces *Hit*/$\overline{miss}$ es 0, informando a la memoria principal que debe entregar la palabra a la CPU e informando a la CPU de que la palabra se retrasará.

Tamaño de línea

Hasta ahora hemos supuesto que cada entrada en la caché consiste en una etiqueta y una única palabra. En las cachés reales debe explotarse la localidad espacial, con lo que en cada entrada de la caché se incluyen además palabras cercanas a la direccionada. Entonces, cada vez que ocurre un *cache miss*, en vez de traerse de la memoria principal una única palabra se trae un bloque de l palabras denominado línea. El número de palabras en una línea es una potencia entera de 2 y las palabras están convenientemente alineadas. Por ejemplo, si las líneas están formadas por 4 palabras entonces las direcciones de las palabras de una línea sólo difieren en los bits 2 y 3. El uso de bloques de palabras cambia la forma en que la caché divide las direcciones en campos. La nueva estructura de campos se muestra en la Figura 14-8(a). Los bits 2 y 3, el campo de palabra, se emplean para direccionar la palabra dentro de la línea. En este caso se emplean dos bits, puesto que hay 4 palabras por línea. El siguiente campo, índice, identifica el conjunto. Aquí hay 2 bits, de modo que hay 4 conjuntos de etiquetas y líneas. Los bits restantes de la dirección de la palabra son el campo de etiqueta, que contiene los restantes 4 bits de la dirección de 10 bits.

La estructura resultante para la caché se muestra en la Figura 14-8(b). La memoria de etiquetas tiene 8 entradas, dos en cada uno de los 4 conjuntos. Para cada entrada de etiqueta existe una línea de 4 palabras de datos. Para asegurar un funcionamiento rápido el índice se aplica a la



□ **FIGURA 14-8**
Caché asociativa por conjuntos con líneas de 4 palabras

memoria de etiquetas para leer simultáneamente dos etiquetas, una para cada conjunto de entradas. A la vez, el índice y la dirección de palabra se aplican para leer dos palabras de la memoria caché de datos que se corresponden con las etiquetas. Una lógica de comparación para cada conjunto compara las etiquetas con la dirección aportada por la CPU. Si ocurre un acierto la palabra de datos correspondiente, ya leída, es colocada en el bus de memoria de la CPU. En caso contrario se señala un *cache miss* y la palabra buscada es entregada por la memoria principal a la CPU. La línea conteniendo la palabra y su etiqueta es también cargada en la caché. Para facilitar la carga de líneas completas la anchura del bus de memoria entre la caché y la memoria principal, y también el camino de datos de la caché, tiene una anchura mayor que una palabra. Idealmente, para nuestro ejemplo, el camino tendría una anchura de $4 \cdot 32 = 128$ bits. Esto permite que una línea completa pueda ser colocada en la caché en un único ciclo de lectura de la memoria principal. Si el camino es más estrecho entonces se necesitaría una secuencia de varias lecturas de la memoria principal.

Otra decisión que el diseñador de la caché debe tomar es el tamaño de la línea. Un bus ancho puede afectar al coste y a las prestaciones y uno estrecho puede ralentizar la transferencia de una línea a la caché. Estas circunstancias aconsejan emplear un tamaño de línea pequeño, pero la localidad espacial aconseja lo contrario. En los sistemas actuales, basados en DRAM síncrona, es fácil leer o escribir líneas grandes sin los problemas de coste y prestaciones comentados. Las rápidas operaciones de lectura y escritura en memoria de palabras consecutivas conseguidas por la memoria DRAM síncrona casan bien con la necesidad de transferir largas líneas de caché.

Carga de la caché

Antes de que cualquier etiqueta o palabra haya sido cargada en la caché todas sus posiciones contienen información inválida. Si en este tiempo se produce un *cache hit* entonces la palabra buscada y enviada a la CPU no provendrá de la memoria principal y será inválida. Según se van buscando líneas en la memoria principal y van siendo cargadas en la caché las entradas de la caché van siendo válidas, pero no existe ningún mecanismo que permita determinar qué entradas son válidas y cuales no. Para solventar este problema se añade a cada entrada de la caché, junto con la etiqueta, un bit de validez. El bit de validez indica que la línea asociada de la caché es válida (1) o inválida (0). Es leído junto con la etiqueta. Si el bit de validez es 0, entonces se produce un *cache miss*, incluso cuando la etiqueta coincida con la dirección entregada por la CPU, de forma que el dato será leído desde la memoria principal.

Métodos de escritura

Hasta ahora nos hemos centrado en la lectura de instrucciones y datos de la caché. ¿Qué sucede cuando ocurre una escritura? Recordemos que, hasta ahora, las palabras en la caché han sido simplemente copias de las palabras de la memoria principal que se leen de la memoria caché para conseguir accesos más rápidos. Pero ahora, que estamos considerando la escritura de resultados, este punto de vista debe cambiar. Las 3 posibles acciones a realizar cuando se desea realizar una escritura son:

1. Escribir el resultado en memoria principal
2. Escribir el resultado en la caché
3. Escribir el resultado en la caché y en la memoria principal

Los métodos empleados en la práctica realizan una o varias de estas acciones. Estos métodos se pueden dividir en 2 categorías principales: *write-through* y *write-back* [5].

En *write-through* el resultado se escribe siempre en la memoria principal. Esto necesita el tiempo de acceso a la memoria principal y puede ralentizar el procesamiento. Esta ralentización puede ser paliada en cierta medida mediante el mecanismo de buffering, una técnica en la que el dato a escribir y su dirección son almacenados por la CPU en registros especiales, denominados buffers de escritura, de forma que la CPU pueda continuar el procesado durante la operación de escritura en memoria principal. En la mayoría de los diseños de cachés el resultado también se escribe en la caché si la palabra ya estaba allí —es decir, si se produce un *cache hit*.

En el método *write-back*, también llamado *copy-back*, la CPU realiza la escritura en la caché sólo cuando se produce un *cache hit*. Si hay un *cache miss*, la CPU realiza la escritura en la memoria principal. Existen dos posibles diseños para el caso de que se produzca un *cache miss*. Uno consiste en leer la línea que contiene la palabra a escribir en la memoria principal en la caché, con la nueva palabra escrita tanto en la caché como en la memoria principal. Esto se denomina *write-allocate*. Esto se hace en la esperanza de que se realicen nuevas escrituras sobre el mismo bloque, lo que dará lugar a *cache hits*, evitando ulteriores escrituras en la memoria principal. En lo que sigue supondremos que se emplea *write-allocate*.

El objetivo de una caché *write-back* es ser capaz de escribir a la velocidad de la caché cuando se produce un *cache hit*. Esto evitará que todas las escrituras se realicen a la velocidad, lenta, de la memoria principal. Además se reducirá el número de accesos a la memoria principal, haciéndola más accesible a DMA, a un procesador de entrada/salida o a otra CPU del sistema. Una desventaja del método *write-back* es que las entradas de la memoria principal correspondientes a palabras de la caché que han sido escritas son inválidas. Desgraciadamente esto puede causar un problema con los procesadores de entrada/salida u otras CPUs que estén accediendo a la misma memoria principal debido a datos «caducados» (*stale*) en la memoria principal.

La implementación del concepto *write-back* requiere una operación de escritura pospuesta (*write-back*) de la posición de la memoria caché que vaya a servir para almacenar una nueva línea de la memoria principal cuando se produce un *cache miss* en lectura. Si la posición en la caché contiene una palabra que haya sido escrita, entonces toda la línea de la caché debe ser escrita en la memoria principal para así liberar esa posición para una nueva línea. Esta escritura necesita un tiempo adicional cuando se produce un *cache miss* en lectura. Para evitar que se produzca esta operación de escritura en cada *cache miss* de lectura se añade un nuevo bit a cada entrada de la caché. Este bit, denominado *dirty bit* [6], que está a 1 si la línea de la caché ha sido escrita y a 0 en caso contrario. La operación de escritura de la línea de la caché en la memoria principal sólo se efectúa si el *dirty bit* vale 1. Si se emplea *write-allocate* en una caché *write-back* entonces la operación de escritura de la línea de la caché en la memoria principal también deberá efectuarse cuando se produzca un *cache miss* de escritura.

Existen otros muchos aspectos a considerar a la hora de elegir los parámetros de diseño de una caché, especialmente para cachés que van a trabajar en sistemas en los que la memoria principal puede ser leída o escrita por dispositivos distintos a la CPU a la que sirve la caché.

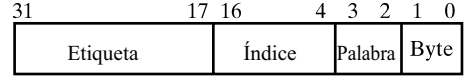
Integración de conceptos

Ahora integraremos todos los conceptos examinados para obtener el diagrama de bloques de una caché asociativa por conjuntos, de 2 vías, *write-through* de 256 kB. Las direcciones de me-

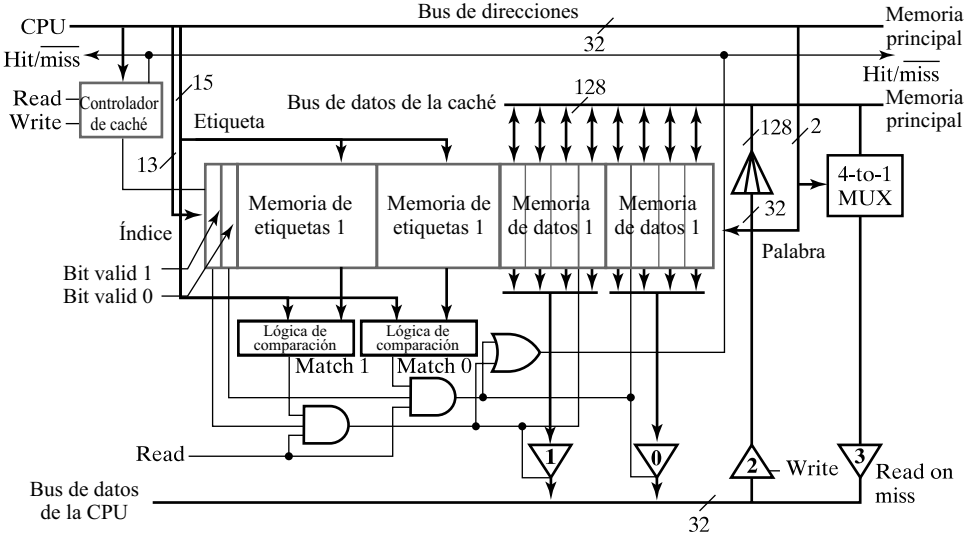
[5] *N. del T.:* Las expresiones inglesas *write-through* y *write-back* no suelen ser traducidas al español.
[6] *N. del T.:* A veces se emplea en español la expresión «bit sucio».

moria mostradas en la Figura 14-9(a) son de 32 bits y emplean direccionamiento de bytes, siendo la longitud de línea $l = 16$ bytes. El índice contiene 12 bits. Dado que se emplean 4 bits para direccionar palabras y bytes y 13 bits para el índice, la etiqueta contiene los 15 bits restantes de la dirección. Cada entrada de la caché contiene 16 bytes de datos, una etiqueta de 15 bits y un bit de validez. La estrategia de reemplazo es aleatoria.

La Figura 14-9(b) presenta el diagrama de bloques de la caché. Puesto que la caché es de 2 vías existen 2 memorias de datos y otras 2 de etiquetas. Cada una de estas memorias contienen $2^{13} = 8192$ entradas. Cada entrada en la memoria de datos contiene 16 bytes. Puesto que se emplean palabras de 32 bits hay 4 palabras en cada entrada de datos de la caché. De este modo, cada memoria de datos consiste en $8192 \cdot 32$ memorias en paralelo con el índice como bus de direcciones. Con el fin de leer una única palabra de estas 4 memorias cuando se produce un *cache hit*, un multiplexor 4 a 1 que emplea las salidas de 3 estados de las memorias selecciona la palabra en base a los 2 bits del campo de palabra de la dirección. Las dos memorias de etiquetas son de $8192 \cdot 15$, y además hay un bit de validez asociado a cada entrada de la caché. Estos bits son almacenados en una memoria de $8192 \cdot 2$ y leídos durante cada acceso a la caché junto con los datos y las etiquetas. Note que el camino de datos entre la caché y la memoria principal es de 128 bits. Esto nos permite presumir que una línea entera podrá ser leída de la memoria principal en un único ciclo de lectura, una suposición que no tiene por qué ser cierta en la práctica. Para comprender los elementos de la caché y cómo trabajan en conjunto estudiaremos los 2 posibles casos de lectura y escritura. Para cada uno de estos casos supondremos que la dirección de la CPU es $0F3F4024_{16}$. Esto da etiqueta = $000011110011111_2 = 079F_{16}$, índice = $1010000000010_2 = 1402_{16}$ y palabra = 01_2.



(a) Dirección de memoria



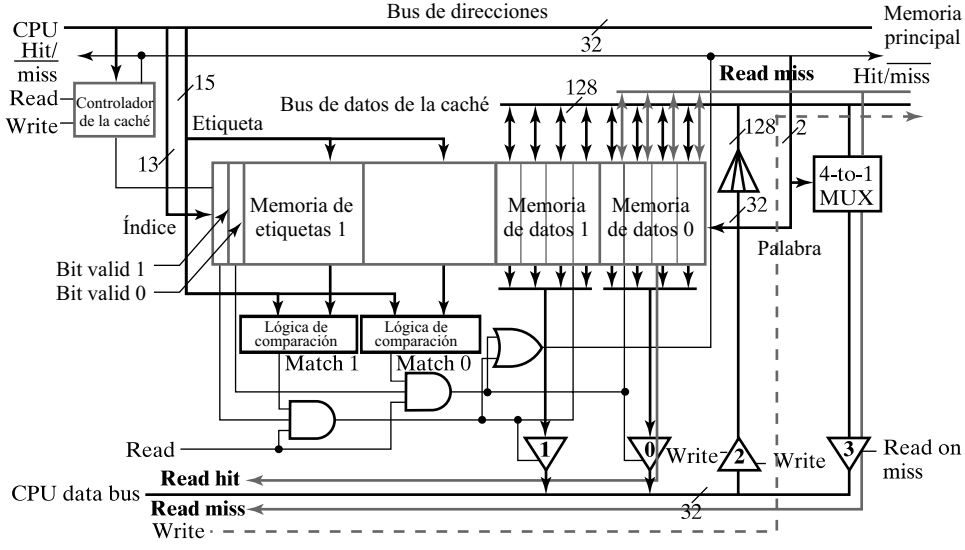
(b) Diagrama de caché

□ **FIGURA 14-9**
Diagrama de bloques detallado de una caché de 256 K

Primero supondremos un *cache hit* de lectura, una operación de lectura en la que la palabra está en una entrada de la caché, como en la Figura 14-10. La caché emplea el campo de índice para leer dos etiquetas de la posición 1402_{16} de las memorias de etiquetas 1 y 0. La lógica de comparación compara las etiquetas de las entradas y en este caso supondremos que la etiqueta 0 casa, haciendo que Match 0 valga 1. Esto no implica necesariamente que haya un *cache hit*, puesto que la entrada puede ser inválida. Entonces se hace la AND de Match 0 con el bit Valid 0 de la posición 1402_{16}. Además el dato puede ser colocado en el bus de la CPU sólo cuando la operación es una lectura. Por ello se hace la AND de Read con el bit Match 0 con el bit Valid 0 para obtener la señal de control de buffer de 3 estados 0. En este caso la señal de control para el buffer 0 es 1. Las memorias de datos han empleado el campo índice para leer ocho palabras de la posición 1406_{16} a la vez que se leían las etiquetas. El campo de palabra selecciona las 2 de estas 8 con palabra $= 01_2$ para colocarlas en el bus de datos que lleva a los buffer de 3 estados 1 y 0. Finalmente, al estar activado el buffer de tres estados 0, la palabra direccionada es colocada en el bus de datos de la CPU. Además la señal envía un 1 a la CPU y a la memoria principal para notificar que ha habido un *cache hit*.

En el segundo caso, también mostrado en la Figura 14-10, suponemos que se produce un *cache miss* de lectura —una operación de lectura en la que la palabra no se encuentra en la caché. Como antes, el campo de índice sirve para leer las etiquetas y los bits de validez, se efectúan dos comparaciones y se chequean dos bits de validez. Para estas 2 entradas ocurre un *cache miss* que es señalizado por *Hit*/$\overline{miss}$ a 0. Esto significa que la palabra debe ser leída de la memoria principal. Por ello, el controlador de caché selecciona la entrada de la caché que será reemplazada y 4 palabras leídas de la memoria principal se aplican simultáneamente al bus de datos de la caché y escritas en la entrada de la caché. A la vez, el multiplexor 4 a 1 selecciona la palabra direccionada por el campo de palabra y la coloca en el bus de datos de la CPU empleando el buffer de 3 estados 3.

En el tercer caso de la Figura 14-10 suponemos una operación de escritura. La palabra de la CPU simplemente es copiada en las 4 posiciones de memoria del bus de datos de 128 bits



□ **FIGURA 14-10**
Caché de 256 K: operaciones de lectura y escritura

de la memoria principal. La dirección en la que debe escribirse esta palabra se aplica al bus de direcciones de la memoria principal para que sea escrita en la posición de la palabra direccionada. Si la dirección causa un *cache hit* entonces el dato es también copiado en la caché.

Cachés de instrucciones y datos

En la mayoría de los diseños de los capítulos anteriores hemos supuesto que era posible buscar una instrucción y leer un operando o escribir un resultado en un sólo ciclo de reloj. Para hacer esto necesitamos una caché que pueda acceder a dos direcciones distintas en un sólo ciclo de reloj. Para responder a esta necesidad discutimos en una subsección anterior el uso de una caché para instrucciones y otra para datos. Esto, además de permitir de forma sencilla el acceso simultáneo a varias direcciones permite que las cachés puedan tener distintos parámetros de diseño. Los parámetros de diseño de cada caché se pueden elegir de forma que se adapten a las características del acceso para búsqueda de instrucciones o para lectura y escritura de datos. Puesto que los requerimientos para estas cachés son usualmente menos estrictos que los de una única caché es posible que puedan ser empleados diseños más simples. Por ejemplo, una única caché puede requerir mapeado selectivo por conjuntos de cuatro vías, mientras que una caché de instrucciones puede necesitar un simple mapeado directo y un caché de datos puede necesitar una estructura asociativa por conjuntos de dos vías.

En otros casos podrá emplearse una única caché para instrucciones y datos. El tamaño de esta caché unificada es usualmente el tamaño combinado de las cachés de instrucciones y datos. La caché unificada permite que sus entradas sean usadas indistintamente para instrucciones y para datos. De este modo, en un instante del tiempo habrá más entradas ocupadas por instrucciones y en otro instante habrá más ocupadas por datos. Esta flexibilidad posibilita el que haya un número mayor de *cache hits*. Este mayor número de *cache hits* puede ser engañoso, puesto que ahora la caché sólo puede manejar un acceso en cada ciclo de reloj, mientras que las cachés separadas permiten dos accesos simultáneos, siempre que uno sea para instrucciones y el otro para datos.

Cachés de múltiples niveles

Es posible extender la profundidad de la jerarquía de memoria añadiendo niveles adicionales de caché. A menudo se emplean dos niveles de caché, referidos como L1 y L2, con L1 el más próximo a la CPU. Para satisfacer las demandas de la CPU de instrucciones y datos se necesita un caché L1 muy rápido. Para alcanzar la velocidad requerida el retardo que se produce al salir de circuito integrado es intolerable. Por ello, el caché L1 se integra, junto con la CPU, en un único circuito integrado, y se denomina caché interna, como en el procesador genérico. Pero el área disponible en un circuito integrado es limitada, de forma que la caché L1 es usualmente pequeña e inadecuada si es la única caché en el sistema. Por ello se añade una caché mayor L2 fuera del circuito integrado del procesador.

El diseño de una caché de dos niveles es más complicado que el de una caché de un sólo nivel. Deben especificarse dos juegos de parámetros. La caché L1 se puede diseñar para unos requerimientos concretos de acceso de la CPU, incluyendo la posibilidad de que haya dos cachés separadas para instrucciones y datos. Además se eliminan las limitaciones en cuanto a pines externos entre la CPU y la caché L1. Además, para permitir lecturas más rápidas el camino entre la CPU y la caché L1 puede ser muy ancho, lo que permitirá la búsqueda simultánea de varias instrucciones. Por otro lado, la caché L2 ocupa el lugar de una caché externa. Sin embar-

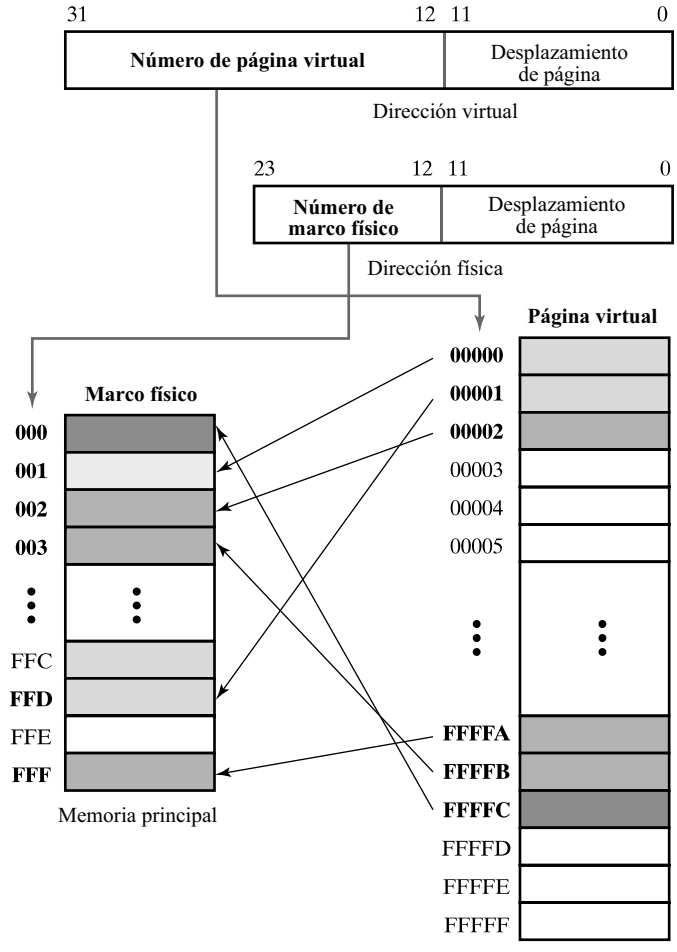
go, difiere de una caché típica en que en vez de proveer datos e instrucciones a la CPU lo hace a la caché de primer nivel L1. Dado que la caché L2 sólo es accedida cuando se produce un *cache miss* en L1 sus patrones de acceso son considerablemente distintos de los generados por la CPU, con lo que sus parámetros de diseño también lo son.

14-4 MEMORIA VIRTUAL

En nuestra búsqueda de una memoria grande y rápida, hemos conseguido la apariencia de una memoria rápida y de tamaño medio mediante el uso de una caché. Con el fin de conseguir la apariencia de una memoria grande exploraremos ahora la relación entre la memoria principal y el disco duro. Dada la complejidad del manejo de las transferencias entre estos dos medios su control requerirá el uso de estructuras de datos y programas. Inicialmente discutiremos las estructuras de datos básicas involucradas y el hardware y software necesarios. Posteriormente nos ocuparemos del hardware específico necesario para efectuar algunas operaciones críticas en cuanto a su velocidad.

Con respecto al tamaño de la memoria, no sólo queremos que el espacio de direcciones virtual parezca que es la memoria principal, sino que en la mayoría de los casos también queremos que este espacio parezca accesible a cada programa que se está ejecutando. Así cada programa «verá» un tamaño de memoria igual a la totalidad de espacio de direcciones virtual. Igualmente importante es para el programador el hecho de que el espacio de direcciones real en la memoria principal y las direcciones reales en el disco son reemplazados por un único espacio de direcciones que no tienen ninguna restricción en su uso. Con esta disposición la memoria virtual no sólo sirve para dar la apariencia de una memoria grande, sino que además libera al programador de la necesidad de considerar las ubicaciones reales de los datos en la memoria principal y en el disco duro. El trabajo del software y del hardware que implementan la memoria virtual es mapear cada dirección virtual, para cada programa, en direcciones físicas en la memoria principal. Además, con un espacio de direcciones virtual para cada programa, que dos direcciones virtuales distintas de dos programas distintos apunten a la misma dirección física. Esto permite compartir datos entre varios programas, reduciendo de esta forma los requerimientos en cuanto a tamaño de memoria principal y tamaño del disco.

Para permitir al software mapear direcciones virtuales en direcciones físicas y para facilitar la transferencia de información entre la memoria principal y el disco duro, el espacio de direcciones virtual se divide en bloques de direcciones, normalmente de tamaño fijo. Estos bloques, denominados páginas, son mayores que las líneas de una caché, pero análogos a ellas. El espacio de direcciones físicas se divide en bloques denominados marcos de página (*page frames*) que son del mismo tamaño que las páginas. Cuando una página está presente en el espacio de direcciones físicas lo está ocupando un marco de página. A modo de ejemplo supondremos que una página está formada por 4 K bytes (1 K palabras de 32 bits). Además supondremos que las direcciones virtuales son de 32 bits. Existe un máximo de 2^{20} páginas en el espacio de direcciones virtuales y, suponiendo una memoria principal de 16 M bytes, existen 2^{12} marcos de página en la memoria principal. La Figura 14-11 muestra los campos de una dirección virtual y una física. La porción de la dirección virtual empleada para direcciones palabras o bytes en una página es el desplazamiento de página (*page offset*), y es la única parte de la dirección compartida entre las direcciones virtuales y las físicas. Note que se supone que las palabras están alineadas con respecto a las direcciones de byte, de modo que la dirección de una palabra siempre termina, en binario, en 00. De la misma forma, las páginas se supone que están alineadas con respecto a las direcciones de byte, de modo que el desplazamiento de página del primer byte de una



□ **FIGURA 14-11**
Campos de direcciones virtuales y físicas y mapeado

página es 000_{16} y el desplazamiento de página del último byte de un página es FFF_{16}. Los 20 bits de la dirección virtual empleados para seleccionar páginas en el espacio de direcciones virtuales se denominan número de página virtual. Los 12 bits de la dirección física empleados para seleccionar páginas de la memoria principal se denominan número de marco de página. la figura muestra un mapeado hipotético del espacio de direcciones virtuales en el espacio de direcciones físicas. Los números de páginas virtuales y físicas se dan en hexadecimal. Una página virtual puede ser mapeada a cualquier marco de página. Se muestran 6 mapeados de páginas de memoria virtual a memoria física. Estas páginas constituyen un total de 24 K bytes. Note que no existen páginas virtuales mapeadas a los marcos de página FFC ni FFE. Así, cualquier dato presente en estas páginas es inválido.

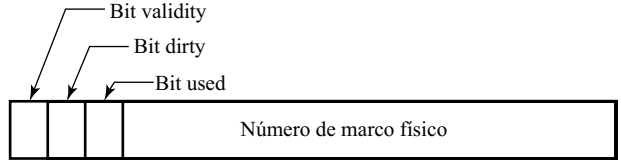
Tablas de páginas

En general existirán un número muy elevado de páginas virtuales, cada una de las cuales debe ser mapeada a memoria principal o al disco duro. Este mapeado es almacenado en una estructura

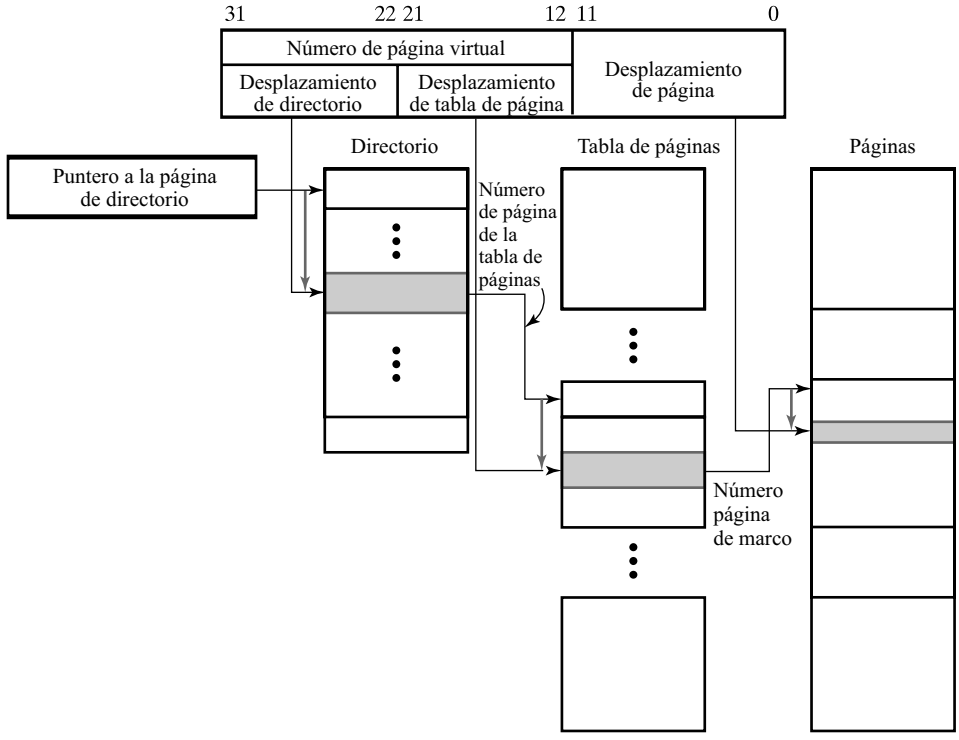
de datos denominada la tabla de páginas. Existen muchas formas de construir la tabla de páginas y de acceder a ella; nosotros asumiremos que esta tabla se almacena en páginas. Suponiendo que una palabra puede representar el mapeado de una página, una página de 4 KB puede almacenar 2^{10}, o 1 K, mapeados. Así el mapeado concreto para el espacio de direcciones de un programa de 2^{22} bytes (4 MB) puede ser almacenado en una página de 4 KB. Una tabla especial para cada programa, denominada la página directorio, contiene el mapeado para ubicar las tablas de páginas de 4 KB.

En la Figura 14-12 se muestra un formato sencillo para una entrada de la tabla de páginas. Doce bits se emplean para designar el marco de página en el que se encuentra ubicada la página en memoria principal. Además existen tres campos de un único bit cada uno. *Valid*, *Dirty* y *Used*. Si *Valid* es 1 entonces el marco de página es válido, siendo inválido en caso contrario. Si *Dirty* es 1 entonces ha ocurrido al menos una escritura en la página desde que fue colocada en la memoria principal, no habiendo existido escrituras en caso contrario. Note que los bits *Valid* y *Dirty* se corresponden con los homónimos de la caché *write-back*. Cuando es necesario eliminar una página de la memoria principal y su bit *Dirty* es 1 entonces la página debe ser copiada en el disco duro. Si el bit *Dirty* es 0 entonces la nueva página que va a ocupar el marco de página es simplemente copiada sobre la página existente. Esto puede hacerse puesto que la versión existente en el disco de la página que va a sobrescribirse sigue siendo correcta. Esto puede hacerse porque el software toma nota en algún sitio de la localización de la página en el disco cuando carga la página en la memoria principal. El bit *Used* sirve para implementar una aproximación burda del mecanismo de reemplazo LRU. Algunos otros bits de la entrada de la tabla de páginas pueden reservarse para almacenar flags usados por el sistema operativo. Por ejemplo, algunos flags pueden indicar protecciones de lectura y escritura de la página cuando ésta es accedida en modo usuario o en modo supervisor.

La estructura de la tabla de páginas que acabamos de describir se muestra en la Figura 14-13. El puntero a la página directorio es un registro que apunta a la posición de la página directorio en memoria principal. la página directorio contienen la posición de hasta 1 K tablas de página asociadas al programa que se está ejecutando. Estas tablas de página pueden estar en la memoria principal o en el disco duro. La tabla de páginas a acceder se obtiene de los 10 MSBs del número de página virtual, que se denominan desplazamiento de directorio (*directory offset*). Suponiendo que la tabla de páginas seleccionada está en memoria principal, ésta puede ser accedida por su número de página. Los 10 LSBs del número de página virtual, que llamaremos desplazamiento de tabla de página (*page table offset*) pueden emplearse para acceder a la entrada de la página que desea alcanzarse. Si la página se encuentra en memoria principal, el desplazamiento de página se emplea para localizar la dirección física de la palabra o byte buscado. Si la tabla de páginas o la página buscada no se encuentran en memoria principal entonces debe ser buscada por el software en el disco duro y colocada en memoria principal antes de que pueda ser accedida. Compruebe que la combinación de desplazamientos con registros o con entradas de las tablas se hace simplemente mediante yuxtaposición, en vez de mediante sumas. Esto no requiere retardos, mientras que la suma originaría retardos importantes.



□ **FIGURA 14-12**
Formato de las entradas en la tabla de páginas



□ **FIGURA 14-13**
Ejemplo de estructura de la tabla de páginas

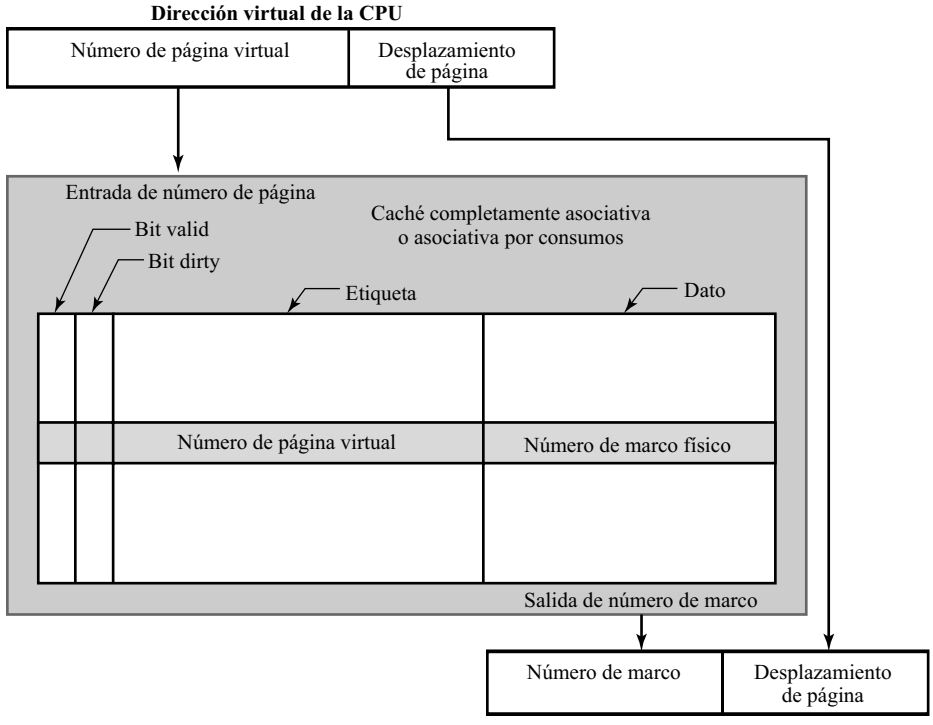
Translation Lookaside Buffer[7]

De la discusión precedente notamos qua la memoria virtual supone una importante merma en las prestaciones del sistema, incluso en el caso ideal de que el directorio, la tabla de páginas y la página a acceder se encuentren en memoria principal. Para el enfoque supuesto se necesitan tres accesos sucesivos a la meoria principal para acceder a un sólo operando o instrucción:

1. Acceder a la entrada en el directorio
2. Acceder a la entrada de la tabla de páginas
3. Acceder al operando o instrucción

Véase que estos accesos se realizan automáticamente por el hardware que es parte de la MMU de la computadora genérica. Así, para conseguir que la memoria virtual sea factible es necesario reducir drásticamente los accesos a memoria principal. Si dispusiésemos de una caché y si todas las entradas estuviesen en la caché entonces el tiempo de cada acceso se reduciría. Aun así serían necesarios 3 accesos a esta cache. Para reducir el número de accesos emplearemos una caché más dedicada a traducir las direcciones virtuales en direcciones físicas. Esta caché se denomina translation lookaside buffer (TLB). Contiene las posiciones de las páginas accedidas más recientemente para así acelerar el acceso a la caché o a la memoria principal. la Figura 14-14 muestra en ejemplo de un TLB, que normalmente es completamente asociativo o asociativo por conjuntos, puesto que necesario comparar el número de página virtual de la CPU

[7] *N. del T.:* Esta expresión no suele traducirse al español.



□ **FIGURA 14-14**
Ejemplo de Translation Lookaside Buffer

con varias etiquetas de números de páginas virtuales. Además de esto, cada entrada de la caché incluye el número de página física para aquellas páginas que se encuentran en la memoria principal y un bit de validez. Si la página se encuentra en memoria principal también se incluye un bit *Dirty*. El bit *Dirty* es análogo para una página en memoria principal que el discutido previamente para una línea de la cache.

Estudiaremos brevemente un acceso a memoria empleando el TLB de la Figura 14-14. El número de página virtual se aplica a la entrada de números de página de la cache. Dentro de la caché este número de página se compara simultáneamente con todas las etiquetas de números de página virtuales. Si alguna comparación es positiva y el bit *Valid* está a 1, entonces ha ocurrido un *TLB hit* y el número de marco de página aparece en la salida de número de página de la caché. Esta operación puede realizarse muy rápidamente y genera la dirección física necesaria para acceder a la memoria o a la caché. Por otro lado, si hay un *TLB miss*, entonces será necesario acceder a la memoria principal para leer la entrada en el directorio y en la tabla de páginas. Si la página física está en la memoria principal entonces la entrada en la tabla de páginas se introduce en el TLB reemplazando alguna de las entradas existentes. En total se necesitan 3 accesos a memoria, incluyendo el acceso al operando. Si la página física no se encuentra en la memoria principal entonces ocurre un fallo de página (*page fault*). En este caso el software se encarga de traer la página desde el disco duro a la memoria principal. Durante el tiempo necesario para realizar esta operación la CPU puede ejecutar un programa diferente, mejor que esperar a que la página sea almacenada en la memoria principal.

Considerando la jerarquía previa de acciones, basada en la presentación de una dirección virtual, vemos que la efectividad de la memoria virtual depende en la localidad espacial y tem-

poral. La respuesta más rápida ocurre cuando el número de página virtual se encuentra en el TLB. Si el hardware es suficientemente rápido y hay un *cache hit* el operando puede estar disponible en tan sólo 1 o 2 ciclos de reloj de la CPU. Esto será tanto más frecuente si a lo largo del tiempo se tiende a acceder a las mismas páginas virtuales. Debido al tamaño de las páginas, si un operando de una página es accedido entonces, debido a la localidad espacial, es probable que otros operandos de la misma página también lo sean. Debido a la capacidad limitada del TLB, la siguiente acción más rápida requiere 3 accesos a la memoria principal y ralentiza el procesado considerablemente. En la peor de las situaciones la tabla de páginas y la página a acceder no se encuentran en la memoria principal. Entonces se requiere la lenta transferencia de 2 páginas —la tabla de páginas y la página desde el disco duro—.

Véase que el hardware básico para implementar la memoria virtual, el TLB, y otras características opcionales para el acceso a memoria se incluyen en la MMU de la computadora genérica. Entre estas características se encuentra el soporte hardware para un nivel adicional de direccionamiento virtual, denominado segmentación, y mecanismos de protección que permitan el adecuado aislamiento y compartición de programas y datos.

Memoria virtual y caché

Aunque hemos considerado la caché y la memoria virtual separadamente, en un sistema real los más probable es que las dos estén presentes. En este caso, la dirección virtual es convertida en una dirección física, y ésta es aplicada a la cache. Suponiendo que el TLB emplea un ciclo de reloj y la caché otro, en el mejor de los casos la búsqueda de una instrucción o de un dato requiere dos ciclos de reloj de la CPU. En consecuencia, en muchos diseños de CPUs segmentadas se permiten 2 o más ciclos de reloj para la búsqueda de operandos. Dado que las direcciones de búsqueda de instrucciones son mucho más predecibles que las de datos, es posible modificar la segmentación de la CPU y considerar el TLB y la caché como otro segmento de dos etapas, de modo que la búsqueda de una instrucción aparenta consumir un único ciclo de reloj.

14-5 RESUMEN DEL CAPÍTULO

En este capítulo hemos examinado los componentes de la jerarquía de memoria. Dos conceptos fundamentales de esta jerarquía con la memoria caché y la memoria virtual.

Basándose en el concepto de localidad de referencias, una caché en una memoria pequeña y rápida que almacena los operandos e instrucciones que, con mayor probabilidad, van a ser usados por la CPU. Normalmente una caché aparenta ser una memoria del tamaño de la memoria principal pero con la velocidad próxima a la de la caché. Una caché trabaja comparando la porción de la etiqueta de la dirección de la CPU con las etiquetas de las direcciones de los datos en la caché. Si una comparación es satisfactoria, y se verifican otros requisitos, entonces ocurre un *cache hit* y el dato puede ser obtenido de la caché. Si ocurre un *cache miss* entonces el dato debe ser obtenido de la memoria principal, más lenta. El diseñador de la caché debe determinar los valores de un número de parámetros, incluyendo el mapeado de las direcciones de memoria principal a direcciones de la caché, la selección de la línea de la caché a ser reemplazada cuando se añade una nueva línea, el tamaño de la caché, el tamaño de las líneas y el método para efectuar escrituras en memoria. Puede haber más de una caché en la jerarquía de memoria, e instrucciones y datos pueden tener cachés separadas.

La memoria virtual se emplea para aparentar una memoria grande —mucho más grande que la memoria principal— con una velocidad que es, en media, parecida a la de la memoria princi-

pal. La mayoría del espacio de memoria virtual se encuentra, en realidad, en el disco duro. Para facilitar el movimiento de información entre la memoria y el disco, ambos se encuentran divididos en bloques de direcciones de tamaño fijo, denominados marcos de página y páginas respectivamente. Cuando una página se carga en la memoria principal, su dirección virtual debe traducirse a una dirección física. La traducción se realiza empleando una o más tablas de páginas. Con el fin de efectuar la traducción en cada acceso a memoria sin generar una gran pérdida de prestaciones se recurre a hardware especial. Este hardware, denominado *Translation Lookaside Buffer* (TLB), es una caché especial que forma parte de la unidad de manejo de memoria (*Memory Management Unit*, MMU) de la computadora.

Juntos la memoria principal, la caché y el TLB producen la ilusión de que se dispone de una memoria rápida y grande, cuando lo que se tiene es, de hecho, una jerarquía de memorias de distintas capacidades, velocidades y tecnologías, con hardware y software encargado de realizar automáticamente las transferencias entre los distintos niveles.

REFERENCIAS

1. MANO, M. M.: *Computer Engineering*: *Hardware Design*. *Englewood Cliffs, NJ*: *Prentice Hall*, 1988.

2. HENNESSY, J. L., and D. A. PATTERSON: *Computer Architecture: A Quantitative Approach.* San Francisco, CA: Morgan Kaufmann, 1996.

3. BARON, R. J., and L. HIGBIE: *Computer Architecture*. Reading, MA: Addison-Wesley, 1992.

4. HANDY, J.: *Cache Memory Book*. San Diego: Academic Press, 1993.

5. MANO, M. M.: *Computer System Architecture*, 3rd Ed. Englewood Cliffs, NJ: Prentice Hall, 1993.

6. PATTERSON, D. A., and J. L. HENNESSY: *Computer Organization and Design: The Hardware/Software Interface.* San Francisco, CA: Morgan Kaufmann, 1998.

7. WYANT, G., and T. HAMMERSTROM: *How Microprocessors Work*. Emeryville, CA: Ziff-Davis Press, 1994.

8. MESSMER, H. P.: *The Indispensable PC Hardware Book*, 2nd ed. Wokingham, U.K.: Addison-Wesley, 1995.

PROBLEMAS

El símbolo (+) indica problemas más avanzados y el asterisco (*) indica que la solución se puede encontrar en el sitio web del libro: http://www.librosite.net/Mano.

14-1. *Una CPU genera la siguiente secuencia de direcciones de lectura en hexadecimal:

54, 58, 104, 5C, 108, 60, F0, 64, 54, 58, 10C, 5C, 110, 60, F0, 64

Suponiendo que al comienzo la caché está vacía y que se utiliza un esquema de reemplazo LRU, determine si cada dirección produce un *cache hit* o *miss* para cada uno de los siguientes casos: **(a)** caché de mapeado directo de la Figura 14-3, **(b)** caché completamente asociativa de la Figura 14-4 y **(c)** caché asociativa por conjuntos de 2 vías de la Figura 14-6.

14-2. Repita el Problema 14-1 para la siguiente secuencia de direcciones de lectura:

0, 4, 8, 12, 14, 1A, 1C, 26, 28, 2E, 30, 36, 38, 3E, 40, 46, 48, 4E, 50, 56, 58, 5E

14-3. Repita el Problema 14-1 para la siguiente secuencia de direcciones de lectura:

20, 04, 28, 60, 20, 04, 28, 4C, 10, 6C, 70, 10, 60, 70

14-4. *Una computadora tiene un espacio de direcciones de 32 bits y una caché de mapeado directo. Se realiza direccionamiento a nivel de byte. La caché tiene una capacidad de 1 K byte y emplea líneas de 32 bytes. Emplea *write-through* y, por tanto, no requiere *dirty bit*.

(a) ¿Cuántos bits tienen los índices de la caché?
(b) ¿Cuántos bits tienen las etiquetas de la caché?
(c) ¿Cuál es el número total de bits almacenados en la caché, incluyendo los bit de validez, las etiquetas y las líneas?

14-5. Una caché asociativa por conjuntos de 2 vías de un sistema con direcciones de 24 bits tiene 2 palabras de 4 bytes en cada línea y una capacidad de 512 K byte. Se realiza direccionamiento a nivel de byte.

(a) ¿Cuantos bits hay en el índice y en la etiqueta?
(b) Indique el valor del índice (en hexadecimal) para las entradas en la caché correspondientes a las direcciones de memoria (en hexadecimal): 82AF82, 14AC89, 48CF0F y 3ACF01.
(c) Pueden estar simultáneamente en la caché todas las entradas de la parte (b).

14-6. *Discuta las ventajas y desventajas de:

(a) Caché separada para instrucciones y datos frente a caché unificada para ambos.
(b) Una caché *write-back* frente a una *write-through*.

14-7. Dé un ejemplo de una secuencia de acceso de instrucciones y datos que arroje una tasa elevada de *cache hits* para cachés separadas de instrucciones y datos y una tasa baja para una caché unificada. Suponga cachés de mapeado directo con los parámetros de la Figura 14-3. Tanto las instrucciones como los datos son palabras de 32 bits y la resolución del direccionamiento es de bytes.

14-8. Dé un ejemplo de una secuencia de acceso de instrucciones y datos que arroje una tasa elevada de *cache hits* para una caché unificada y una tasa baja para cachés separadas de instrucciones y datos. Suponga que las cachés de instrucciones y datos son asociativas por conjuntos de 2 vías y que la caché unificada en una asociativa por conjuntos de 4 vías, ambas con los parámetros de la Figura 14-6. Tanto las instrucciones como los datos son palabras de 32 bits y la resolución del direccionamiento es de bytes.

14-9. Explique por qué *write-allocate* no se emplea normalmente en las caché *write-through*.

14-10. Una estación de trabajo de alta velocidad tiene palabras de 64 bits y direcciones de 64 bits con capacidad de direccionar bytes.

(a) ¿Cuántas palabras caben en el espacio de direcciones de la estación?
(b) Suponiendo una caché de mapeado directo con 8192 líneas de 32 bytes ¿cuántos bits hay en cada uno de los siguientes campos de las direcciones en la caché: (1) Byte, (2) índice y (3) etiqueta?

14-11. *Una caché tiene un tiempo de acceso desde la CPU de 4 ns, y la memoria principal lo tiene de 40 ns. ¿Cuál es el tiempo de acceso efectivo para la jerarquía caché-memoria principal si la fracción de *cache hits* es: **(a)** 0.91, **(b)** 0.82 y **(c)** 0.96?

14-12. Rediseñe la caché de la Figura 14-17 de modo que mantenga la misma capacidad, pero tenga 4 vías en vez de 2.

14-13. +Se quiere rediseñar la caché de la Figura 14-9 de forma que sea *write-back* con *write-allocate* en vez de *write-through*. Realice las siguientes tareas asegurando que considera todos los posibles problemas relacionados con la operación *write-back*.

(a) Dibuje un nuevo diagrama de bloques.
(b) Explique la secuencia de acciones que propone cuando se produce un *cache miss* de lectura y otro de escritura.

14-14. *Un sistema de memoria virtual emplea páginas de 4 K byte, palabras de 64 bits y direcciones virtuales de 48 bits. Un programa concreto y sus datos necesitan 4263 páginas.

(a) ¿Cuál es el número mínimo de tablas de página necesarias?
(b) ¿Cuál es el número mínimo de entradas necesarias en la página directorio?
(c) A la vista de las respuestas dadas en (a) y (b) ¿cuántas entradas habrá en la última tabla de páginas?

14-15. Un pequeño TLB tiene las siguientes entradas para un número de páginas virtual de 20 bits de longitud, un número de página física de 12 bits y un desplazamiento de página de 12 bits

Bit Valid	Bit Dirty	Etiqueta (número de página virtual)	Dato (número de página física)
1	1	01AF4	FFF
0	0	0E45F	E03
0	0	012FF	2F0
1	0	01A37	788
1	0	02BB4	45C
0	1	03CA0	657

Los números de página y los desplazamientos están en hexadecimal. Para cada dirección virtual listada indique si ocurre un TLB *hit* y, en caso afirmativo, indique la dirección física: **(a)** 02BB4A65, **(b)** 0E45FB32, **(c)** 0D34E9DC y **(d)** 03CA0777.

14-16. Una computadora dispone de un máximo de 384 M byte de memoria principal. Trabaja con palabras de 32 bit, direcciones virtuales de 32 bit y emplea páginas de 4 K byte. El TLB contiene entradas que almacenan los bits *Valid*, *Dirty* y *Used*, el número de página virtual y el número de página física. Suponiendo que el TLB es completamente asociativo y que tiene 32 entradas responda a las siguientes cuestiones:

(a) ¿Cuántos bits de memoria asociativa necesita el TLB?
(b) ¿Cuántos bits de SRAM necesita el TLB?

14-17. Cuatro programas se ejecutan concurrentemente en una computadora multitarea con páginas de memoria virtual de 4 K byte. Cada entrada en la tabla de páginas es de 32 bit. ¿Cuál es el número mínimo de bytes de memoria principal ocupados por las páginas directorio y las tablas de páginas para los 4 programas, si el número de páginas de cada programa, en decimal, es de 6321, 7777, 9602 y 3853?

14-18. *En las caché se pueden usar los enfoques *write-through* y *write-back* para manejar las escrituras. Pero para el caso de memoria virtual sólo se emplea un enfoque similar al *write-back*. Explique por qué.

14-19. Explique claramente por qué los conceptos de memoria caché y memoria virtual no serían útiles si no se verificase la localidad de referencia en los patrones de direccionamiento de las memorias.

ÍNDICE

Números

Complemento a 1, 200
Complemento a 2,198, 200
AOI 2-1, 68
AOI 3-2-2, 69

A

Acceso directo a memoria (DMA). *Véase* también DMA, 569, 576-579
  Cesión de bus, 576
  Controlador, 576-578
  Petición, 578-579
  Petición de bus, 576-577
  Ráfaga de transferencia, 576
  Reconocimiento, 579
  Robo de ciclo, 576
  Transferencia, 578-579
  Transferencia en un solo ciclo, 576
Adición de ceros, 12
Álgebra Booleana, 28, 31-38
  Identidades básicas, 33-35
  Principio de dualidad, 37
Algoritmo de máquinas de estados, 345-349
  Consideraciones temporales, 347
  Diagrama, 345-348
    Ejemplos, 348-354
  Diseñar, 353
Almacenamiento
  Asíncrono, 229
  Elementos, 229
  Recursos, 427-428
Almacenar en FIFO. *Véase* almacenar First-in, first-out
Almacenar First-in, first-out (FIFO), 597
ALU. *Véase* unidad aritmético-lógica
Amplitud relativa, 95
AND-OR-INVERT (AOI), 68
Anticipación de datos, 526-528
AO. *Véase* AND-OR,
AOI. *Véase* AND-OR-INVERT,
Arquitectura, 460
  Carga/almacenamiento, 437, 465
  CISC, 474
  Conjunto de instrucciones, 427, 460, 474-476
  Direccionamiento, 465-469
  Memoria a memoria, 465-467
  Memoria-registro, 466
  Pila, 467
  Recientes innovaciones, 545-546
  Registro a registro, 465-467
  RISC, 475, 513
  Un solo acumulador, 466
  Varias unidades de ejecución, 542
Arquitectura CISC, 474, 495
Arquitectura de conjunto de instrucciones, 408, 427, 439, 474-476
  RISC, 513-517
Arquitectura de la CPU:
  Segmentada, 509-542
  Superescalar, 542, 544
  Supersegmentada, 543
Arquitectura RISC, 475, 495
Array de lógica programable (PAL), 119
  Dispositivos, 122-124
  Implementación de circuito combinacional, 159-161
Array lógico iterativo, 190

Array lógico programable (PLA), 119, 121-122
Arrays de puertas programables en campo, 124, 164
ASM. *Véase* Algoritmo de máquinas de estados

B

Banco de registros, 419
Banco de test, 88
Banderas. *Véase* códigos de condición
Barrel shifter, 417-419, 518
Basado en puertas de transmisión, 73-74
Base, 8
Base 8, 7
BCD, 17-19, 24
Bifurcación retardada, 528
Big-endian, 297
Binario, 4
  Multiplicación, 13
  Resta, 13, 197-202
  Suma, 12, 190-197
Bit, 5
Bit de entrada, 304
Bit de paridad, 19-24
Bit saliente, 305
Bit válido, 600
Bloque ASM, 346
Bloque funcional, 85
Bloque reutilizable, 85
Bloques, 83
Bloques predefinidos, 84-85
Bloques primitivos, 84, 86
Boole, George, 28
Bottom-up design, 86
Bucle de espera, 569
Buffer, 66
Buffering, 601
Burbuja, 66, 523
Bus, 325-328
  Basado en multiplexor, 325-328
  Triestado, 326-328
Bus de entrada/salida (E/S), 7
Bus multiplexor, 325-327
Bus universal serie (USB), 566
Byte, 22, 378

C

Cabeza, 553
Caché, 591-605
Caché de datos, 604
Caché de instrucciones, 604
Caché externa, 6, 604
Caché interna, 6
Caché unificada, 604
CAD. *Véase* Diseño asistido por computadora
Caja de decisión escalar, 346
Caja de decisión, 346
Caja de salida condicional, 346
Caja de vector de decisión, 346
Canal de datos, 558
Captura de esquemáticos, 85-87
Caracteres de control de comunicación, 22
Carga, 292
Carga especulativa, 545
Celda:
  Dinámica, 393, 396-403
    Síncrona, 398-402
  Direcciones de las columnas, 395
  Direcciones de las filas, 495
  Doble Tasa de Transferencia de Datos, 402-403
  Estática, 383-392
  Lectura destructiva, 394
  Posición de forma aleatoria, 596
  RAMBUS (RDRAM), 398, 402-404
  Restaurar, 394
  Temporización, 395-398
  Tipos, 398-403
Chequeo de Redundancia Cíclica (CRC), 568
Chip. *Véase* Circuitos integrados
Ciclo de diseño, 97-103
Cilindro, 553
Circuito aritmético, 411-415
Circuito con dos niveles, 43
  Criterio de coste, 45-46
  Mapa de cuatro variables, 51-54
  Mapa de dos variables, 46-47
  Mapa de tres variables, 47-51
  Optimización, 44-54
Circuito secuencial, 227-290
Circuito secuencial asíncrono, 228
Circuito secuencial máquina de mealy, 246
Circuito secuencial máquina de Moore, 246
Circuito secuencial síncrono, 228
  Asignación de estados, 259-261
  Asíncronos, 228
  Definiciones, 228-230
  Diagrama de estados, 247-248, 253-260
  Diseño con flip-flops D, 259
  Diseño, 252-267
  Estado actual, 245
  Estado futuro, 245
  Estados sin utilizar, 261-262
  Modelo de Mealy, 246
  Modelo de Moore, 246
  Procedimiento de análisis, 243-252
  Procedimiento de diseño, 252
  Síncronos, 228, 229

Tabla de estados, 245-247
Temporización, 248-250
Circuitos CMOS (Suplemento), 69
Circuitos combinacionales, 82, 134
Circuitos digitales, 27
Circuitos integrados RAM, 405-411, 415-421
Circuitos integrados, 28, 91-92
Circuitos lógicos combinacionales, 27-79
Definidos, 32
Circuitos lógicos, 82, 414
Circuitos, integrados, 27, 86, 90-92
Clear, 241
Codificación, 143-147
Codificadores, 143-147
Expansión, 146-147
Prioridad, 145-147
Código ASCII, 22-23, 24
Caracteres de control, 22-23
Códig Gray, 19-21, 24, 47
Binario reflectado, 21
Código binario de n bits, 17, 139
Código binario, 17
Código de condición, 461
Código de marca, 553
Código de ruptura, 553
Código estandarizado americano para intercambio de información. *Véase* ASCII
Código exceso-3, 98
Código Gray binario reflectado, 21
Código scan, 565
Códigos alfanuméricos, 21-23
Códigos de scan-K, 553
Códigos Hamming, 404
Collapsing, 62
Complememto a 1, 200
Complememto a la base, 200
Complemento a 2, 198, 200
Complemento a la base menos 1, 200
Complemento, de una función, 37-38
Complementos, 200-202
Resta con, 200-202
Comunicación:
Full-duplex, 563
Semi-duplex, 563
Serie, 562-569
Simples, 563
Tiempo de respuesta, 563
Concadenación, 352
Condiciones de indiferencia, 59-60
Configuración separada de E/S, 578, 558
Conflicto de datos, 524-528
Conjunto de instrucciones, 427
Conjunto de registros, 461
Contador, 317-319
Contador asíncrono, 311-315
Contador de programa (PC), 344, 426
Contadores, 292, 311-320
Ascendente-descedente, 315
Asíncrono, 311-315
BCD, 317-318
Binario, 311-317
Con carga paralela, 316-317
Con puertas en paralelo, 315
Con puertas en serie, 315
Dividido por N, 317
Módulo N, 317
Paralelo, 314
Secuencia arbitraria, 318-320
Serie, 315
Serie-paralelo, 315
Síncrono, 314-320
Contadores binarios, 311-318
Contracción de lógica, 209, 210-215
Control:
Cableado, 354-363
CISC, 530-542
Microprogramado, 368-371, 537-543
RISC, 519-520
Segmentado, 509-512
Único ciclo, 433-441
Varios ciclos, 441-452
Control de conflictos, 527-530
Control microprogramado, 368, 537-542
Organización, 370
Control segmentado, 509-512
Conflicto de datos, 320-528
Control de conflictos, 527-530
Diagrama de ejecución de patrones, 508
Parada, 524
Relleno, 508-509
Vaciado, 509
Controlador de disco, 7, 554
Controlador de DRAM, 404
Conversión:
Base r a decimal, 8-10
Binario a decimal, 9
Binario a hexadecimal, 10
Binario a octal, 10-12
Decimal a base r,15
Decimal a binario,15-16
Decimal a octal, 15
Fracciones decimal a binario, 16
Fracciones decimales a octal, 16-18
Lógica positiva a lógica negativa, 95-96
Octal a hexadecimal, 12
Conversión de números. *Véase* Conversión
Conversor de código BCD a exceso-3, 98-100
Conversores de código, 98
Copy-back, 601
Bit de validez, 600

Carga, 600
Dato, 604
Dirty bit, 601
Etiqueta, 592
Externa, 587, 604
Hit, 593
Índice, 592
Instrucción, 604
Interna, 604
L1, 604-605
L2, 604-605
Línea, 597
Mapeado, 594-598
Mecanismo de buffering, 601
Métodos de escritura, 600-601
Miss, 594
Múltiples niveles, 604-605
Tamaño de conjuntos, 597
Tamaño de línea, 599-601
Unificada, 604
Write-allocate, 601
Write-back, 601
Write-through, 600-601
Coste por literal, 45
CPU de alto rendimiento, conceptos, 542-545
CPU super segmentada, 543
CPU. *Véase* unidad central de proceso (CPU)
CRC. *Véase* chequeo de redundancia cíclica
Cuadrados adyacentes, 47-51
Cuadrados, 46-47

D

Decimal codificado en binario. *Véase* BCD
Decimal:
Aritmética, 215
Códigos, 17-20
Decodificación, 139-144
Implementación de circuitos combinacionales, 151
Decodificador, 139
Decodificador BCD a siete segmentos, 101-104
Decodificador de *n* a *m* líneas, 139
Decremento, 213-214
Demultiplexor, 143
Descomposición, 62
Descripción estructural, 87
Desplazadores, 416-419
Combinacional, 417-419, 518
Desplazamiento de directorio, 607
Desplazamiento, 304
Bit entrante, 304
Bit saliente, 305
Detección de error:
Paridad, 19-20
Y códigos de corrección (suplemento), 404
Diagrama de tiempos, 30
Diagrama del circuito, 32
Dígito más significativo, 8
Dígito menos significativo (lsd), 8
Dirección, 379
Absoluta, 514
Efectiva, 468
Etiqueta, 590
Explicita, 462
FOCET, 430
Implícita, 462
Dirección física, 605
Dirección indirecta, 447
Dirección relativa, 514
Dirección vectorizada, 572
Dirección virtual, 605
Direccionamiento de bytes, 588
Direccionamiento del operando, 462-468
Direccionamiento, de bytes, 588
Disco duro, 5-7, 553-555
Formato, 554
Retardo de giro, 554
Tasa de transferencia del disco, 554
Tiempo de acceso, 554
Tiempo de búsqueda, 554
Diseño asistido por computadora, 86-87
Diseño de logica combinacional, 80-131
Diseño jerárquico, 83-86
Diseño Top-down, 81, 82, 86
Diseño VLSI:
Arrays de puertas, 104
Células estándar, 104
Full-custom, 104
Disipación de potencia, 92
Display de 7 segmentos, 101
Dispositivo PAL. *Véase* array lógico programable
Dispositivos:
Alta escala de integración (LSI), 91
Media escala de integración (MSI), 86, 91
Muy gran escala de integración, 91
Pequeña escala de integración, 91
Dispositivos de alta escala de integración (LSI), 91
Dispositivos de mediana escala de integración (MSI), 91
Dispositivos de muy alta escala de integración (VLSI), 91, 124
Dispositivos de pequeña escala de integración (SSI), 91
Distancia dos, uso del término, 70
Distribución, 143
División por una constante, 213
DMA. *Véase* acceso directo a memoria
DRAM. *Véase* RAM. Dinámica
Dual:
De una ecuación, 36
De una expresión algebraica, 33

E

E2PROM (borrable eléctricamente, ROM programable), 121
E/S. *Véase* Entrada/salida
  Bus, 7
    Serie, 565-569
  Comandos, 580-581
  Entradas de selección de registro, 558
  Procesadores, 570, 579-582
  Puertos, 478, 558
  Registro de control, 558-559
  Tasa de trasnferencia, 556
  Unidades de interfaz, 556-562
E/S ubicada en memoria, 557
Efectos de formato, 22
Eliminación, 62
Ensamblador, 431
Entrada/salida, 7
  Asíncrono, 558-559
  Independiente, 478
  Interfaces, 556-562
  Procesadores, 558, 570, 579-582
  Puerto, 478, 558
  Separada, 478
  Ubicada en memoria, 478
Entradas de selección de registro, 558
Entradas directas, 241
EPROM (borrable, ROM programable, 120-121
Equivalencia. *Véase* NOR exclusiva
Espacio-tiempo, 330
Especulación de datos, 545
Esquemático, 83
Estado, 228, 245
  Caja, 345
  Diagrama, 247-248
  Máquina, 345
  Tabla, 245-247
    Entrada, 245
    Estado actual, 245
    Estado futuro, 245
    Salida, 245
Estado actual, 245
Estado de alta impedancia, 71
Estado Hi-Z. *Véase* Estado de alta impedancia
Estándar:
  Carga, 93
  Formas, 39-44
Estados sin utilizar, 261-262
Excepción, 494
Expansión, 140-143
Exponente, 483
  Sesgado, 485-486
Expresión Booleanas, 31, 37
Extensión, 214
  De signo, 215, 430, 514
Extracción, 62

F

Factorización, 62
  Algebraica, 62
Factorización algebraica, 62
Fan-in, 92
Fan-out, 91, 93-95
  Circuito libre, 111
  Puntos, 113
Flanco, 92-94
Flattening, 62
Flip-flop, 229, 230, 234-244, 264-268
  Características:
    Ecuación, 265
    Tabla, 265
  Clear, 241
  Disparado por flanco, 235, 238
  Disparado por pulso, 238
  Ecuaciones de entrada de los Flip-flop, 243
  Ecuaciones de entrada, 243-244
  Indicador de salida pospuesto, 239
  Indicador dinámico, 240
  JK, 265-267
  Maestro-esclavo, 235-238
  No transparente, 235
  Preset, 241
  Reset asíncrono, 241
  Set asíncrono, 241
  Símbolos Gráficos Standard, 239-242
  T, 265-267
  Tabla de excitación, 265
  Tiempo de hola, 242
  Tiempo de setup, 242
  Tiempos de los Flip-Flops, 242-244
  Tiempos de retardo de propagación, 241
  Trigger, 235
Flip-flop disparado por flanco, 235, 238
Flip-flop JK, 265-267
Flip-flop maestro-esclavo, 236-238
Flip-flop T, 265-267
Flip-flop tipo D, 238-240
  Con habilitación, 294
FPLA, 122
FPU. *Véase* Unidad en punto flotante (FPU)
Fracción, 483
Frecuencia, reloj, 248
Función Booleana, 31
  Incompletamente especificada, 59
  Salida múltiple, 31
  Única salida, 31

Función de generación, 196
Grupo, 197
Función impar, 70-71
Función par, 70-72
Función propagación, 196
Grupo, 197
Funciones de varios bits, 135-137
Funciones especificadas incompletamente, 59
Funciones lógicas básicas, 134-139

G

G (Giga), 9

H

Habilitación, 137-139
Handshaking, 561-562
HDL:
Análisis, 88
Descripción estructural, 87
Elaboración, 88
Inicialización, 88
Representación del multiplicador, 363-365
Simulación, 88
Testbench, 88

I

IC. *Véase* Circuitos integrados (IC)
IC RAM, array de, 389-392, 404
Identidades, 68
Identificador de paquete (PID), 567-568
Implementación de funciones combinacionales:
Empleando Decodificadores, 152-155
Empleando Memorias de Sólo Lectura, 157-159
Empleando multiplexores, 154-156
Empleando Tablas de Búsqueda, 163-165
Usando Arrays de Lógica Programable, 171-174
Usando Arrays Lógicos Programables, 159-161
Implicante, 54
Implicante primo esencial, 54-57
Implicante primo no esencial, 56
Implicantes primos, 55-57
Esencial, 54-56
No esencial, 56
Regla de selección, 56
Incremento, 211-212
Indicador de negación, 66
Indicador de polaridad, 96
Indicador de salida postpuesta, 239
Instancia, 85
Instanciación, 55
Instrucción, 344, 426-427
Accesos, 443
Campos, 421, 426
Código de operación, 428
Ejecución, 427, 445
Formatos, 428-431
Registro, 442-443
Instrucciones:
Aritmética, 479, 484-485
Bifurcación condicional, 488-491
Bifurcación y salto, 487
Cero direcciones, 464-466
Control de programa, 487-495
Desplazamiento, 481-482
Dos direcciones, 463
Llamada y retorno de subrutina, 488-491
Manipulación de datos, 479-482
Manipulación lógica y de bit, 480-482
Pila, 476-478
Punto flotante, 483-487
Transferencia de datos, 476-479
Tres direcciones, 462-464
Una dirección, 463-465
Instrucciones de cero direcciones, 464-466
Integración, niveles de, 92
Interfaz de bus, 7
Interrupción, 493-495
Daisy chain, 573-574
Externa, 493-495
Interna, 493
No vectorizada, 572
Paralela, 574-576
Prioridad, 572-576
Procesamiento de externas, 494-495
Sectorizada, 572
Software, 493-494
Interrupción de programa, 492-495
Interrupción no vectorizadad, 572
Interrupción vectorizada, 572
Inversor 33. *Véase* también puerta NOT
Invertida, 135
Iterativo:
Array lógico, 190, 330
Circuitos combinacionales, 190-191

J

Jerarquía, 83

K

K (Kilo), 9

L

Latch D, 233-235
Latches, 230-234
  D con puertas de transmisión, 234
  D, 233-234
  Estado RESET, 231
  Estado SET, 231
  Símbolos gráficos estándar, 239-241
  SR, 231-233
    Con entrada de control, 233
  Transparente, 235
Lectura destructiva, 394
Lenguaje ensamblador, 460
Lenguaje máquina, 460
Leyes asociativas, 33
Leyes conmutativas, 33
Leyes distributivas, 34
Librería, 104
Librería de células, 104-105
LIFO. *Véase* Pila Last-in, first-out (LIFO)
Línea, 599
Líneas de barrido, 555
Literal, 36-37
Little-endian, 297
Localidad de Referencia, 590-592
  Espacial, 590
  Temporal, 590
Localidad espacial, 590
Localidad temporal, 590
Lógica binaria, 28-29
Lógica compartida, 305
Lógica dedicada, 305
Lógica negativa, 95-97
  Indicador de polaridad, 96
Lógica positiva, 95-97
LRU. *Véase* usado menos recientemente (LRU)
LSI. *Véase* Dispositivos, alta escala de
LUT. *Véase* tabla de búsqueda (LUT)

M

M (Mega), 9
Manipulación algebraica, 35-38
Mantisa. *Véase* fracción o significando
Mapa, 44-60
  Cuatro variables, 51-54
  Dos variables, 46-47
  Manipulación, 54-60
  Tres variables, 47-51
Mapa de fusibles, 118
Mapa de Karnaugh. *Véase* Mapa
Mapa K. *Véase* Mapa
Mapeado asociativo por conjuntos, 597
Mapeado completamente asociativo, 594-594
Mapeado de la caché, 594-599
  Asociativo por conjuntos, 597
  Completamente asociativo, 594-595
  Directo, 594-595
Mapeo:
  Asociativo por conjuntos de *s* vías, 597
  Completamente Asociativo, 593-594
  Directo, 593-594
Mapeo directo, 593-594
Mapeo tecnológico, 89, 104-113
Margen de ruido, 92
Matriz de muestro, 552
Maxitérmino, 39-43
Memoria, 7, 379-406
  Asociativa, 595
  Definiciones, 378
  Dinámica, 383
  Estática, 383
  No volátil, 383
  Virtual, 605-610
  Volátil, 383
Memoria asociativa, 595
Memoria Caché. *Véase* también Caché, 592
Memoria de acceso aleatorio (RAM), 7, 377, 378-383
Memoria de control, 368
  De escritura, 368
Memoria de sólo lectura (ROM), 119-121, 378
  Borrable eléctricamente, programable (EEPROM), 121
  Borrable, programable (EPROM), 120-121
  Implementando un Circuito Combinacional, 158-159
  programable (PROM), 118-120
Memoria programable de sólo lectura (PROM), 118-120
Memoria virtual, 592, 605-610
Memorias flash, 118
Microinstrucción, 368
Microoperaciones, 296-297, 299-305
  Aritméticas, 299, 300-302
  Desplazamiento, 299, 304-305
  Lógicas, 299, 301-305
  Transferencias, 299
Microprograma, 368, 539-542
Minitérmino, 39-43
MMU. *Véase* unidad de manejo de memoria (MMU)
Modem, 562
Modos de direccionamiento, 468-474
  Directo, 470-471
  Implícito, 469
  Indexado, 472-473
  Indirecto, 471
  Inmediato, 469

Registro base, 473
Registro indirecto, 469-470
Registro, 469-470
Relativo, 472
RISC, 516
MSI. *Véase* Dispositivos, mediana escala de integración
Multiplexor, 147-150
Expansión, 149-150
Puerta de transmisión, 151
Triestado, 151
Multiplicación:
Algoritmo, 349-350
Binaria, 13, 208-209
Octal, 14
Por una constante, 213
Multiplicadores, 208-209, 349-363
Control, 358-362
Control con Registro de Secuencia y Descodificador, 357-359
Representación en Verilog, 365-368
Representación en VHDL, 363-365
Ruta de datos, 350
Un flip-flop por estado
MUX, 149

N

NaN. *Véase* No es un número,
Netlist, 87
No es un número, 481
No retorno a cero invertida (NRZI), 567
NOR exclusiva, 69
Notación polaca inversa (NPI), 467
Notación posfija. *Véase* notación polaca inversa
NRZI. *Véase* No retorno a cero invertida
Número de página virtual, 606
Número normalizado en punto flotante, 483
Números:
Binario, 9-10
Binario con signo, 203-205
Punto flotante normalizado, 483
Números binarios con signo, 203-205
Números binarios, 9-11
Complemento a 1, 200
Complemento a 2, 198, 200
Con signo, 203-205
Signo y complememto, 204
Signo y magnitud, 203
Sin signo, 198, 200-202
Números hexadecimales, 10-12, 24
Suma, 13-14
Números octales, 10-12, 24
Multiplicación, 14

O

OA. *Véase* OR-AND
OAI. Vere OR-AND-INVERT
Op code. *Véase* código de operación
Operación:
Código, 428
Mnemónico, 431
Operación AND, 28, 104
Operación de complemento, 28
Operación de escritura, 378, 380
Operación de lectura, 378, 380
Operación NOT, 28
Operación OR, 28, 66
Operaciones aritméticas, 12-18, 24
Conversión de decimal a otras bases, 15-17
Operaciones de transferencia de registro, 295, 297-299
Operaciones elementales, 296
Operaciones en punto flotante, 6, 483
Operaciones lógicas, 28
Operando inmediato, 430
Optimización de circuitos multinivel, 61-65
Optimización de producto de sumas, 57-58
Optimización del mapa, 44-60
OR - exclusiva, 67, 69-72
Or exclusiva, 74
OR-AND (OA), 69
OR-AND-INVERT (OAI), 69
Ordenador genérico, 5-8
Organización del procesador, 460
Organización super-escalar, 542, 544
Overflow, 206-208

P

Página, 591, 605
Desplazamiento, 605
Fallo, 609
Marcos, 605
Número de marco, 606
Número de página, 607
Tabla, 606-607
Desplazamiento, 607
Página directorio, 606-608
Palabra, 378
Palabra de control, 422-426
Parada de conflicto de bifurcación, 528
Parada por conflicto de datos, 524
Parte Significativa, 486
Patrón de sincronización (SYNC), 567
Pila Last-in, first-out (LIFO), 464
Pila, 464
Arquitectura, 466

Instrucciones, 464, 468
Puntero (SP), 461
Pista, 553
PLA programable en campo (FPLA), 122
PLA. *Véase* Array lógico programable, 121-122.
Posición de Caché:
Aleatorio, 596
FIFO, 597
Usado menos recientemente (LRU), 597
Potencia de dos, 9
Predicación, 545
Predicción de bifurcación, 529
Preset, 241
Principio de dualidad, 37
Procesador de comunicación de datos, 579
Procesador de conjunto de instrucciones complejo, 474, 530-543
Procesador de conjunto de instrucciones reducido, 475
Procesador, 6
Arquitectura, 408, 468
Ciclo básico, 461
CISC. *Véase* procesador, complejo
Comunicación de datos, 579
Conjunto de instrucciones
Conjunto de instrucciones complejo, 474, 530-543
Diseño, 3, 407-458
E/S, 579
Estructura, 5
Microprogramado, 537
Registro de status, 461
Retardo en el peor caso, 439-440
Segmentado, 508-513
Único ciclo, 433-440
Varios ciclos, 441-452
Procesadores digitales, 3-5, 6
Producto de maxitérminos, 41
Producto de sumas, 44
Programa, 427
Programación de PLD
Antifusible, 117
Borrable eléctricamente, 118
Borrable, 118
Fusible, 117
Máscara, 117
PROM. *Véase* Memoria programable de solo lectura (PROM)
Protección de violación, 493
Puerta, 29-31, 65-69
Compleja, 66, 68
Coste por entrada, 45-46
Primitiva, 66-67
Tipos, 65-69
Transmisión, 73-74
Universal, 66
Puerta NAND, 66, 68, 104
Puerta NOR, 66
Puerta universal, 66
Puertas compuestas, 66, 68
Puertas de transmisión, 73-74
Puertas lógicas, 28, 29-31, 65-69
Símbolos, 67
Puertas primitivas, 66-67
Puntero a la página directorio, 607
Punto de la base, 8
Punto flotante, 483-487
Exponente sesgado, 485-486
Formato estándar de los operandos, 485-487
Números, 482, 483
Operaciones aritméticas, 484-485
Puntos, fan-out, 113

R

RAM. *Véase* Memoria de acceso aleatorio
Rango de los números, 12
Realización del control:
Registro de Secuencia y Descodificador, 357-359
Un Flip-flop por Estado, 358-363
Rectángulos, 48-50
Rendimiento, pipeline, 511-513
Redondeo del acarreo, 201
Refresco, 396-398
Contador, 397
Controlador, 397
Iniciación, 396-398
Operación, 398
Temporización, 396
Tipos, 398
Refresco de la memoria DRAM, 383, 398
Regiones de transición, 30
Registro, 292
Carga paralelo, 293
Carga, 292
Célula, 319
Desplazamiento, 307-311
Diseño de una célula, 319-325
Registro compuesto, 352
Registro de control de datos, 368
Registro de control de dirección, 368
Registro de secuencia y decodificador, 357-359
Registros de desplazamiento, 307-311
Bidireccional, 310-311
Con carga paralela, 308-310
Unidireccional, 310
Regla de selección, 56
Relleno, ceros, 214-215
Relleno de ceros, 213-215
Reloj:
Anchura, 242-244

Circuitos secuenciales síncronos, 229-230
Frecuencia, 248
Gating, 241, 294
Generador, 229
Periodo, 248
Pulso, 229
Skew, 241, 250
Transición, 238
Reset asíncrono, 241
Resta:
Binaria, 13
Con signo utilizando complementos, 204
Sin signo utilizando complementos, 201
Restaurar, 394
Retardo de giro, 554
Retardo de propagación, 91-95, 96
Retardo de transporte, 93
Retardo inercial, 93
Retardo: Delay:
Inercial, 93
Reducción de retardo, transformación para, 65
Transporte, 93
ROM. *Véase* Memoria de solo lectura
Ruta de datos, 6, 291, 395, 408-411
RISC, 516-519
Segmentada, 504-512
Simulación, 425

S

Salidas en alta impedandia, 71-74
Sector, 553-554
Secuenciador, 369
Secuenciamiento y control, 344-375
Segmentación, 610
Selección, 147-152
Selección combinada, 387
Selección de bit, 384
Selección de columna, 387
Selección de fila, 387
Selección de palabra, 384
Selector de datos, 149
Semisumador, 191
Sentencia condicional:
Forma, 298
Forma, 305
Señal de control, 295
Señales, 4, 215
Separadores de información, 22
Serie:
Contadores, 314
E/S basada en paquetes, 565-569
Memoria, 378
Suma, 329-332
Transferencia, 328-332
Set asíncrono, 241
Siguiente estado, 245
Símbolo AND, 28-29
Símbolo OR, 29
Símbolos gráficos: -:
Flip-flop, 239-242
Latch, 239
Puertas, 30, 65-69
Simplex, 563
Simulación, 250-252
Funcional, 251
Temporal, 251
Simulación lógica:
Flip-flop maestro-esclavo, 236
Latch SR, 232
Simulador lógico, 86
Síntesis lógica, 88-90
Sintetizadores lógicos, 87
Sistema no programable, 344-345
Sistema programable, 344
Sistemas de memorias, 587-614
Sistemas de numeración, 8-12
Sistemas digitales, 4
Relación con el diseño de la CPU, 546-547
Skew, reloj, 241, 250
Soporte, 164
SRAM. *Véase* Memoria estática de acceso aleatorio
SSI. *Véase* Dispositivos, pequeña escala de integración
Strobing, 559-560
Suma, 18-19
BCD, 17-19
Binaria con signo, 204
Binaria, 12
Hexadecimal, 13-14
Serie, 329-332
Suma de miniterminos, 40
Suma de productos, 42-44
Optimización, 42, 45, 51, 54, 56-58, 62, 74
Suma de términos, 39-40
Sumador:
Con acarreo anticipado,194-197
Con acarreo serie,193
Sumador binario con acarreo serie, 193-194
Sumador binario, 190-197
Sumador completo, 191, 192-197
Sumador con acarreo anticipado, 194-197
Sumador con acarreo serie, 193
Sumador-restador,202-208
Sustitución, 62

T

Tabla de búsqueda (LUT), 118, 164

Tabla de verdad, 29
AND, 29
Condensada, 138, 145
Función, 31-32
NOT, 29
Operación, 29
OR, 29
Tamaño de línea, 599-601
Tamaño del conjunto, 597
Tarjeta gráfica, 555-556
Tasa de baudios, 564
Tasa de transferencia del disco, 554
Teclado, 7, 552-553, 564-566
Tecnología flash, 118
Temporización de la memoria, 381-382, 396
Teorema de consenso, 37-38
Teorema de DeMorgan, 34-35, 37-38, 67
General, 34
Término producto, 39
Términos de una expresión, 31
Thrashing, 592
Throughput, 505
Tiempo de acceso, 554
Tiempo de búsqueda, 554
Tiempo de ciclo de escritura, 381
Tiempo de hold, flip-flop disparado por flanco, 242
Tiempo de latencia, 505
Tiempo de propagación de alto a bajo, 92-93
Tiempo de propagación de bajo a alto, 92
Tiempo de rechazo, 93
Tiempo de respuesta, 563
Tiempo de setup, disparado por flanco
Flip-flop, 242
Tiempo del controlador, 554
Tiempos de retardos de propagación:
Flip-flops, 242
Puertas, 91-95
Tipos de memoria, 383
Tira de un bit:
Dinámica, 394-398
Estática, 383
TLB. *Véase* Translation lookaside buffer (TLB)
Tpdv. *Véase* retardo de propagación
tPHL. *Véase* tiempo de propagación de alto a bajo
tPLH. *Véase* tiempo de propagación de bajo a alto
Trama, 555
Transferencia, 135
Basada en multiplexores, 325-327
Basada en triestado, 326-327
Controlada por programa, 570-572
Iniciada por interrupción, 571
Modos, 569-571
Serie, 328-332
Varios registros, 325-327
Transferencia de Bus, 325-328
Transferencia de regsitro, 295-297
Destino, 297
Fuente, 297
Transiciones, 30
Transistor, 4
Translation lookaside buffer (TLB), 608-610
Transmisión:
Asíncrona, 563
Síncrona, 563, 564
Tasa en baudios, 564
Transparente, 235
Trap, 493
Triestado, 66-73
Buffers, 71-73
Bus, 326-327
Trigger, 235
U
Un flip-flop por estado, 358-362
Unicode (suplemento), 22
Unidad aritmético-lógica, 408, 411-417
Unidad Central de Proceso (CPU), 6, 7, 503-550
Unidad de control, 6, 291, 295, 344-345
Unidad de manejo de memoria (MMU), 6, 7, 608
Unidad en punto flotante (FPU), 6, 7, 459
Usado menos recientemente (LRU), 597
USB. *Véase* Bus universal serie

V

Valor constante, 135
Variables binarias, 28
Varias unidades de ejecución en paralelo, 542
Verificación, 113-116
Verilog, 87, 172-178, 218-221, 275-281, 333-335, 365-368, 404
Asignación de bloques, 275
Asignación non-blocking, 275
assign, 176
case, 278
Circuitos secuenciales, 275-281
Comentarios, 173
Concatenación, 219-221
Control de eventos, 276
Default, 280
Descripción de comportamiento, 218-220
Descripción de flujo de datos, 176
Descripción estructural, 174
Directivas de compilador, define, 279
if-else, 276
input, 173
Modelado de un multiplicador, 365
Modelado de un registro de desplazamiento, 331-333
module, 173
Operador @, 275

output, 173-174
process:
always, 275
initial, 275
register, 275
Representación de códigos binarios, 279
Representación de un circuito secuencial, 279
Representación de un contador, 333-335
Símbolos de transferencia de registros, 299
Temporización y reset, 275
vectors, 175
wire, 174
VHDL, 87, 165-173, 215-219, 268-274, 332-334, 363-365, 404
Arquitectura de una entidad, 167
Asignación de estados, 274
attribute, 274
begin, 168
case, 270, 273-274
Codificación de estados, 274
Comentario, 166
Componente, 168
Concatenación, 217-218
Declaración de entidad, 167
Descripción de comportamiento, 217-219
Descripción de flujo de datos, 170
Descripción estructural, 167
end, 168
Generación de almacenamiento, 267
if-then-else, 268-270
library, 167
Lista de sensivilidad, 268
Lógica estándar, 167
Modelado de un contador, 332
Modelado de un multiplicador binario, 363-365
Modelado de un registro de desplazamiento, 331
others, 171, 273
package, 167
Peligro, 273
Peligro, 273
port:
Declaración, 167
map, 168
process, 267-269
Puertas, 174
Representación de un circuito secuencial, 270
Sentencia when-else, 170
Señales, 168
Símbolos de trasferencia de registros, 300
std_logic, 167
std_logic_vectors, 168-170
Tiempo delta, 168
type, 271
use, 167
variable, 268
with-select, 170
VLSI. *Véase* dispositivos, muy alta escala de integración
Voltaje de entrada, 4
Voltaje de salida, 4

W

Write-back, 601
Write-through, 601

X

XOR. *Véase* OR exclusiva